Faster Depth-First Subgraph Matching on GPUs

Lyuheng Yuan*, Da Yan*, Jiao Han*, Akhlaque Ahmad*, Yang Zhou+, Zhe Jiang#
*Indiana University Bloomington {lyyuan, yanda, jiaohan, akahmad}@iu.edu
+Auburn University, USA yangzhou@auburn.edu
#University of Florida, USA zhe.jiang@ufl.edu

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00244

***Abstract*—Subgraph search problems such as maximal clique enumeration and subgraph matching generate a search-space tree which is traversed in depth-first manner by serial backtracking algorithms that are recursive. Since Jenkins et al. reported the backtracking paradigm to be sub-optimal for GPU acceleration, breadth-first traversal of the search-space tree is widely adopted by GPU algorithms. However, they produce a lot of intermediate subgraphs that exhaust the GPU device memory. Recent works revive the depth-first backtracking paradigm for GPU acceleration, where each warp is a basic processing unit with its own stack in device memory for subgraph backtracking. However, they adopt complicated methods for load balancing that incur a lot of overheads. They also use hardcoded fixed space for stacks that is determined ad-hoc and may lead to inaccuracy when the allocated space is insufficient.**

In this paper, we use subgraph matching as a case study to propose novel depth-first GPU solutions to address the above problems. Our approach, called T-DFS, decomposes computation into independent tasks that process search-space subtrees, which are managed by an efficient lock-free circular task queue. Tasks are distributed to different warps for parallel processing, and a novel timeout mechanism is used to eliminate straggler tasks to ensure load balancing. We also support flexible and fine-grained dynamic memory allocation for stack spaces to avoid the stack space allocation pitfalls of existing works. Extensive experiments on real graphs show that T-DFS significantly outperforms existing depth-first GPU solutions for the subgraph matching application.

I. Introduction

Given a graph $G = (V, E)$ where V (resp. E) is the vertex (resp. edge) set, we consider the subgraph search problem that finds (or counts) those subgraphs of G that satisfy certain conditions. These problems have a wide range of applications including social network analysis and biological network investigation [27]. However, the search space is the power set of V: for each subset $S \subseteq V$, we check whether the subgraph of G induced by S satisfies the conditions.

Subgraph search is usually solved by a recursive backtracking algorithm in the serial context, which conducts depth-first search (DFS) on a search-space tree. Examples include Ullmann's algorithm for subgraph matching [45], the Bron-Kerbosch algorithm for enumerating maximal cliques [23] and k-plexes [55], the set-enumeration algorithm for enumerating maximal quasi-cliques [27], [28], [36], [37], and the pivoting algorithm for k-clique counting [32]. Without loss of generality, this paper focuses on subgraph matching.

For simplicity, let us assume that graphs are unlabeled (or equivalently, all vertices/edges have the same label). Fig. [1] illustrates the *state space search tree* [41] of subgraph matching with query graph G_Q against the data graph G, where we match data vertices to query vertices $u_1, u_2, \ldots, u_5$ one at

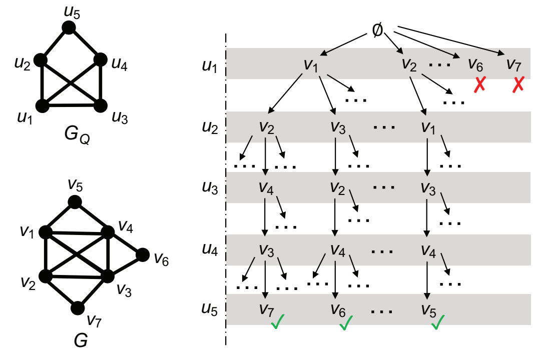

Fig. 1. State Space Tree for Subgraph Matching

a time. Each path starting from the dummy root $\emptyset$ records a partial match $S = [v_{i_1}, v_{i_2}, \ldots, v_{i_\ell}]$ which matches each v_{i_j} to the query vertex u_j. For example, the leftmost path shown $S = [v_1, v_2, v_4, v_3, v_7]$ is a valid match to G_Q (and outputted), while the rightmost path $S = [v_7]$ is pruned since v_7 (with degree 2) cannot match u_1 which has 3 neighbors in G_Q.

Since G_Q is symmetric, a subgraph can be matched several times. For example, $S_1 = [v_1, v_2, v_4, v_3, v_7]$ and $S_2 = [v_4, v_3, v_1, v_2, v_7]$ are two matches to G_Q, but they correspond to the same subgraph of G. To eliminate this redundancy, we can generate constraints between vertices to break the pattern symmetry. In particular, let us denote $id(u_i)$ to be the ID of the matched vertex in G, then we can add the restriction $id(u_1) < id(u_3)$ to G_Q which avoids the enumeration of S_2.

Since subgraph isomorphism is NP-complete, many parallel systems [24]–[26], [33], [38]–[40], [44], [46], [51], [52] have been developed to accelerate its processing, utilizing CPU cores. GPU solutions to subgraph matching have also been proposed, such as GSI [54], cuTS [50], PBE [29], VSGM [35], SGSI [53], and Pangolin [25]. These solutions maintain and grow the intermediately matched subgraphs in the state space tree in breadth-first search (BFS) order (i.e., one level at a time) to allow coalesced memory access.

Since [34] reported the backtracking paradigm to be sub-optimal for GPU acceleration (their maximal clique enumeration on a GPU achieves a performance of only 1.4–2.25× that of a single CPU core), depth-first backtracking algorithms on GPU(s) were not explored until recently, where several works revive this paradigm by treating each warp as a basic processing unit with its own stack in device memory for subgraph backtracking. Among them, STMatch [47] and EGSM [43] copes with subgraph matching, [21] copes with maximal clique enumeration, and [20] copes with k-clique counting.

To create enough tasks for concurrent processing by warps, it is common to treat the first level of the search tree as individual tasks. In Fig. 1, the subtrees under $v_1, v_2, \ldots, v_7$ at the u_1-level are tasks each of which can be traversed in DFS order by a warp. However, the sizes of the subtrees can be very imbalanced since if a matched data vertex has a much higher degree in G, it can create a much larger fanout in the next level in the state space tree. Most DFS solutions above actually treat edges (i.e., the first two levels of the search tree) as individual tasks to mitigate this problem. In Fig. 1, the subtrees under $(v_1, v_2), (v_1, v_3), \ldots, (v_2, v_1)$ at the (u_1, u_2)-level now becomes the tasks. While the tasks are more fine-grained, workload imbalance still exists and existing DFS solutions propose complicated methods for load balancing that incur a lot of overheads, and they use hardcoded fixed space for stacks that is determined ad-hoc and may lead to inaccuracy when the allocated space is insufficient.

In this paper, we use subgraph matching as a case study to propose novel depth-first GPU solutions to address the above problems. Our approach, called T-DFS, uses a novel timeout mechanism to eliminate straggler tasks, where a long-running task will be split into smaller tasks that are added to an efficient lock-free circular task queue Q_{task}. Idle warps prioritize the fetching of tasks from Q_{task} over taking new initial tasks for processing to keep the number of tasks in Q_{task} small. T-DFS also supports fine-grained dynamic memory allocation for stack spaces to avoid the stack space allocation pitfalls of existing works. In summary, our main contributions are:

- We propose a novel timeout mechanism to eliminate straggler tasks for load balancing, which is more efficient than existing heavyweight work stealing approaches.
- We propose an approach to managing tasks by an efficient lock-free circular task queue Q_{task}, and keeping the number of tasks in Q_{task} small.
- We propose an approach to dynamically allocate stack spaces on demand to avoid the stack space allocation pitfalls of existing works.
- We integrate algorithmic optimizations, edge filtering and set intersection result reuse, to improve efficiency.
- Extensive experiments on 12 real graphs are conducted to show that T-DFS significantly outperforms existing depth-first GPU solutions for subgraph matching.

Note that the first three techniques are general for depth-first subgraph search on GPUs, not just limited to our targeted subgraph matching application.

The rest of this paper is organized as follows. Section II presents the preliminaries and reviews the related work. Then, Section III presents our system design, and Section IV reports our experiments. Finally, Section V concludes this paper.

II. Preliminaries and Related Work

This section first introduces the Ullmann's algorithm for subgraph isomorphism, and the various algorithm improvements on top. We then describe works that parallelize subgraph matching, with a special focus on GPU solutions, especially those based on depth-first backtracking search.

Algorithm 1 *subgraph_match*$(G_Q = (V_Q, E_Q), G = (V, E))$

```
1: generate matching order $\pi = [u_1, u_2, \ldots, u_k]$ where $u_i \in V_Q$
   and $k = |V_Q|$
2: enumerate($\emptyset, 1, \pi, G_Q, G$)
   Procedure enumerate($S, i, \pi, G_Q, G$)
3:     $C_S(u_i) \leftarrow$ viable vertex candidates in $G$ to match $u_i$
4:     for each $v \in C_S(u_i)$
5:         Append $S$ with $(u_i, v)$
6:         if $|S| = k$ then output $S$
7:         else enumerate($S, i+1, \pi, G_Q, G$)
8:         Pop $(u_i, v)$ from $S$
```

Ullmann's Algorithm. Algorithm 1 sketches Ullmann's recursive algorithm [45] for subgraph matching. Specifically, Line 1 first selects a favorable vertex matching ordering π. For example, u_1 can be selected as the vertex with the highest degree in G_Q, which has the most edge constraints and tends to match to fewer data vertex candidates. Then, Line 2 calls the recursive procedure *enumerate*(.) to match data vertices to the query vertices in π one at a time.

Here, procedure *enumerate*(S, i, π, G_Q, G) is called with the first $(i-1)$ vertices in π already matched, and this matching is recorded by $S = [(u_1, v_{j_1}), (u_2, v_{j_2}), \ldots, (u_{i-1}, v_{j_{i-1}})]$ (we can simply write $S = [v_{j_1}, \ldots, v_{j_{i-1}}]$ when π is clear from the context). Then, Line 3 obtains a candidate set $C_S(u_i)$ of data vertices to match u_i. Let us denote $B^\pi(u_i) = \{u_j \mid j < i \wedge (u_j, u_i) \in E_Q\}$ as the backward neighbors of u_i in G_Q for given a query vertex ordering π. Then,

$$C_S(u_i) = \bigcap_{u_j \in B^\pi(u_i)} N(S[u_j]) \tag{1}$$

which ensures that all backward edges are matched, where $S[u_j]$ denotes the data vertex matched to u_j in S, and $N(v)$ here refers to those neighbors of v whose labels (if available) match that of u_i. For each candidate $v \in C_S(u_i)$, Line 5 then append (u_i, v) to S and if there are still vertices to match in π, Line 7 recursively calls *enumerate*(.) to continue matching from u_{i+1}. If all vertices are matched, Line 6 outputs S as a valid match to G_Q. Note that before executing Line 5, we also need to make sure v is not already matched to another query vertex in S, which is omitted in our presentation hereafter for simplicity. Note that *enumerate*$(\emptyset, 1, \pi, G_Q, G)$ basically searches the state space tree as illustrated in Fig. 1 by DFS.

To illustrate, the leftmost path in Fig. 1 gives a recursion path $S = [(u_1, v_1), (u_2, v_2), (u_3, v_4), (u_4, v_3), (u_5, v_7)]$. This path succeeds since $|S| = k$ in Line 6.

A number of improvements have been proposed on top of Ullmann's algorithm, which are summarized by [42].

Parallel CPU-Based Systems. We next briefly review the parallel systems for subgraph search.

Arabesque [44], RStream [46] and Fractal [26] extend subgraphs (aka. embeddings) of size i to generate subgraphs of size $(i+1)$, and they aim to support not only subgraph search, but also frequent subgraph pattern mining (FSM). However, since a size-$(i+1)$ subgraph can be generated from different size-i subgraphs, additional graph isomorphism checks are needed to eliminate redundancy. Arabesque [44] and RStream [46] conduct BFS on the search tree, while

Fractal [26] supports DFS in execution to be memory-efficient. Pangolin [25] also conducts BFS for subgraph extension but supports GPU execution. G-thinker [51] and G-Miner [24] only target subgraph search, and they allow users to specify backtracking algorithms directly to avoid materializing intermediate subgraphs as much as possible, while Arabesque, RStream and Pangolin [25] have to materialize intermediate subgraphs, causing enormous memory and computation cost.

GraphPi [40] and GraphZero [38] further narrow down their application scope to subgraph matching enumeration only. Since different vertex matching order π leads to different costs. AutoMine [39], GraphPi and GraphZero adopt a compilation-based approach to generate subgraph enumeration code with a favorable vertex matching order. They also design methods to avoid redundant computation caused by pattern symmetry.

GPU Solutions to Subgraph Matching. Recent works begin to explore the use of GPUs to further accelerate subgraph enumeration/matching. Since Jenkins et al. [34] reported the backtracking paradigm to be sub-optimal for GPU acceleration, earlier works are BFS-based.

To ensure coalesced memory access, each subgraph is extended by a warp to allow coalesced memory access. Specifically, recall from Eq (1) that a warp would need to compute "set intersections" to obtain candidates for subgraph extension. The threads of a warp computes an intersection $A \cap B$ by having each thread check an element $a \in A$ with binary search against B to see if $a \in B$. Since elements of A read by the threads of a warp are consecutively stored, and elements common to A and B are appended to an output buffer using atomicAdd(.) that atomically forwards the write location, the memory access pattern of the warp is coalesced.

Among the BFS solutions, GSI [54] and cuTS [50] maintain and grow the intermediately matched subgraphs in device memory. Since GPU device memory is limited, PBE [29], VSGM [35] and SGSI [53] explore methods to partition a large input graph so that only a partition needs to be loaded to a GPU for processing at each time. Some systems such as VSGM conduct expensive preprocessing (e.g., k-means clustering and bin packing) to partition the input graph, the time of which frequently surpasses the time of computing subgraph matching itself according to our experimental tests.

To prevent device memory overflow, PBE [29] adopts a pipelined approach to memory management. Specifically, before PBE tries to compute candidate vertices on the next level for subgraphs at the current level (by set intersections), it first estimates an upper bound of the number of candidate vertices (e.g., by the smallest set size before set intersection) and cuts the subgraphs into some small batches (that remaining device memory can accommodate the extension) to be processed one at a time. However, after each batch is processed, it needs to release the memory space dedicated for this batch, and allocate new memory space for processing the next batch, which incurs overheads. Moreover, PBE maintains subgraphs as a prefix tree so prior levels need to be kept in device memory, so it aims to make the new level as tight as possible (to leave room for subsequent layers of extension) by computing the next-level subgraphs once to get the exact space needed by the next level for memory allocation, followed by another pass of subgraph computation to populate these subgraphs. This leads to redundant computation overheads.

More recently, DFS solutions to subgraph matching are explored on GPUs by STMatch [47] and EGSM [43], where each warp conducts DFS on its assigned chunk of initial tasks (i.e., state space subtrees) by maintaining its own stack, and load balancing is achieved by work stealing which splits heavy tasks. While STMatch is a pure DFS solution, EGSM advocates a BFS-DFS hybrid solution where the more efficient BFS is used when device memory permits, and if memory becomes insufficient, it falls back to DFS to match the remaining vertices in π. Moreover, G_Q is initially divided into vertex groups, and each extension is by a group (using DFS) rather than by an individual vertex, so as to reduce the BFS levels and the number of intermediate results. EGSM also features the construction of a new Cuckoo trie data structure to index candidates for a given query graph as a preprocessing step, so as to support fast candidate pruning and lookup on the fly during subgraph matching. However, as Fig. 3 of the EGSM paper [43] shows, the structure has three levels so it requires one extra memory access compared to the tyical CSR format, resulting in a higher memory access overhead and hence only paying back if the the pruning is highly effective (e.g., when the label set is large). Also note that while building an index for a query graph to maintain compact candidates is a popular processing step in CPU-based subgraph matching (see Section 3.1 of [42]), we are not aware of another work besides EGSM that aims to extend this technique for GPU-based processing. Most GPU systems just conduct simple degree- and label-based candidate pruning and set intersection result reuse [29]–[31], [40], which we also adopt (please refer to the last part of Section III for the details).

We find, however, that the work-stealing methods of both STMatch and EGSM are inefficient. Specifically, STMatch lets an idle warp i directly probe other warps j for their workloads as illustrated in Fig. 2, which necessitates locking Warp j's stack first. Here, Warp i will find the highest level of Warp j's stack that still has unprocessed vertices, and take half of the remaining vertices in this level to process their state space subtrees. Note that if Warp i does not lock Warp j's stack before stealing in Fig. 2, another Warp k could also steal v_2 and v_3 to its own stack like Warp j, causing v_2 and v_3 to be computed twice by both Warps j and k.

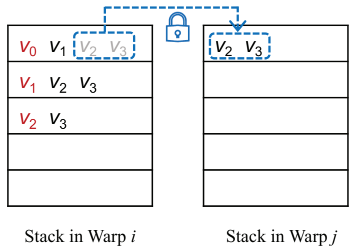

Fig. 2. Half Stealing in STMatch

In STMatch, idle warps first steal tasks from its own block, and if there is no task to steal in the entire block, the first warp of the block then steals tasks from warps of other blocks. Since each stack (of a warp i) is frequently accessed by Warp i itself during DFS-based subgraph matching while may be accessed by any other warp for task stealing, not only the other warps but also Warp i itself need to frequently lock and unlock the stack each time it is accessed, creating a lot of overheads. Even worse, Warp i busy-waits on its stack when another warp is stealing (copying data) from the stack, which causes GPU core underutilization. We found that imposing locks can sometimes deteriorate the overall efficiency. For example, enabling this works-stealing method can take up to more than 3× the time than if no stealing is done (See Section IV-C).

On the other hand, in EGSM, a warp θ detects potential long-running tasks during DFS, and calls a new kernel with the proper number of warps to process the current state space subtree T, while θ itself backtracks to continue processing (and may create more new kernels). For example, when a warp matches to a tree path that creates a fanout of 1024 in the next level, instead of processing the next level by itself, the warp calls a new kernel with 32 warps (each processing 32 vertices in the next level). However, creating new kernels is costly since a new dedicated stack space needs to be allocated before the new kernel starts. Besides, many active kernels may coexist at the same time which adds burden to warp scheduling. It is more desirable to distribute the tasks to existing warps rather than creating a new kernel (T-DFS does this as we shall see).

III. System Design

This section describes the system design of T-DFS. Let us first consider the simple case without load balancing. As Fig. 3 shows, a subgraph matching job needs only one kernel call, where each warp is a basic processing unit with its own stack in the GPU global memory. The data graph G is stored in the device memory using the compressed sparse row (CSR) format, and for simplicity, Fig. 3 assumes that initial tasks are created by vertices of G to match $u_1 \in \pi$. Note that in the actual implementation, we use edges (i.e., the first two levels) to create more fine-grained initial tasks.

If multiple GPUs are used, the initial tasks are first evenly assigned to all the GPUs by round robin. At each GPU, every idle warp will obtain the next available chunk of initial tasks from those that are assigned to the GPU for processing, until no more initial tasks are available. The default chunk size is 8. Fig. 3 illustrates the chunk-to-warp assignment. T-DFS currently does not do task migration among GPUs for simplicity, but we observe that the initial task assignment is reasonably balanced among GPUs to scale out well.

When a warp obtains a chunk of initial tasks, they are put into the topmost level of the warp's stack (see Fig. 3). Afterwards, each warp processes its obtained initial tasks one by one, where each task is processed by DFS with the help of the warp's stack. Note that the stack space can be preallocated in the device memory to be large enough, i.e., having k levels with each level having the capacity to hold d_{max} elements,

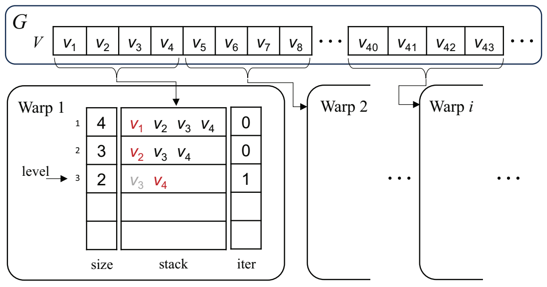

Fig. 3. Initial Tasks & Warps and Their Stacks

where d_{max} is the maximum vertex degree of G (which can hold any intermediate set intersection results). No additional space is required during the DFS processing of the tasks, so this approach uses bounded device memory and is thus scalable. This is in contrast to BFS approaches that need to materialize all intermediate subgraphs in a level of the state space tree which can easily grow beyond the device memory capacity. We next describe the algorithm at each warp.

Stack-Based Warp Algorithm. As Fig. 3 shows, let *level* be the current level during DFS, then the call stack of a warp maintains 3 arrays, where *stack*[*level*] keeps the candidate vertices that match $u_{level} \in G_Q$, *size*[*level*] keeps the number of candidate vertices, and *iter*[*level*] keeps the position of the current vertex in *stack*[*level*]. In Fig. 3, the current DFS path is $[v_1, v_2, v_4]$ as highlighted in red in *stack*[], and v_3 in Level 3 has been processed and is hence marked gray.

Algorithm 2 shows the backtracking algorithm of a warp for processing a chunk of initial tasks, which translates the recursive algorithm described in Algorithm 1 into an iterative one that explicitly manages the recursion stack. Specifically, Line 1 fills the top level of *stack*[] with the next chunk of initial vertices, which are candidates to match u_1. If no more initial vertices are available in Line 1, the warp exits an outer-loop that repeatedly calls Algorithm 2 to process more chunks. We omit this outer-loop for simplicity. Then, the while-loop from Line 2 processes the initial tasks in the current chunk. Specifically, Line 3 checks if all the query vertices have been matched. If so, a valid match is found, so the else-branch

Algorithm 2 Stack-Based Warp Backtracking Algorithm

```
1:  stack[1] ← next chunk of initial tasks $\{v_j\}$,  level ← 1
2:  while true do
3:      if level $< k$ then
4:          if size[level] $= 0$ then
5:              if level = 1 then break
6:               Extend stack[level] along the current path ending at
    Level (level $- 1$)
7:          if iter[level] $<$ size[level] then
8:              level ← level + 1
9:          else
10:             iter[level] ← 0      % Clear stack[level]
11:             if level $> 1$ then
12:                 level ← level $- 1$
13:                 iter[level]++
14:     else
15:         Emit the valid match
16:         level ← level $- 1$
```

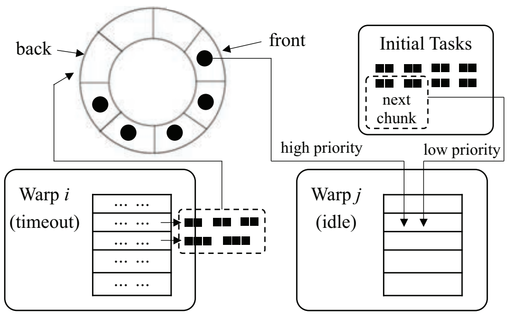

Fig. 4. Load Balancing

from Line 14 is executed with Line 15 emitting this match and Line 16 backtracking to the previous level.

Otherwise, we enter the if-branch from Line 3. If the current level is empty (Line 4), then if this is already the top level (Line 5), we break out of the while-loop from Line 2 since all the assigned initial vertices have been processed; otherwise, Line 6 continues to fill *stack*[*level*] with the candidates computed from Eq (1), where the backward neighbors are tracked by *iter*[]. Now if u_{level} has a match in *stack*[*level*], Line 8 proceeds computation to the next level. While if all candidates in the current level have been exhausted (Line 7), the program will backtrack to the previous level (Lines 9–13).

To illustrate Algorithm 2, refer to Fig. 1 again. Level 1 is initialized by the vertices in G at Line 1 of Algorithm 2. Then, the leftmost path in Fig. 1 is traversed as follows: Level 2 is extended with $N(v_1)$ at Line 6, Level 3 with $N(v_1) \cap N(v_2) = \{v_3, v_4\}$, Level 4 with $N(v_1) \cap N(v_3) = \{v_4\}$, and so on. In contrast, assume that $S = [v_6]$ is expanded with candidates $\emptyset$ in Level 2 for ease of explanation (in reality, v_6 will be pruned by degree filtering rather than expanded), then the warp backtracks at Line 12 to Level 1 and then matches v_7 to u_1.

Load Balancing. Since initial tasks can correspond to state space subtrees of drastically different sizes, a task with a giant state space subtree would become a straggler, so it is desirable to divide its computation into smaller tasks for concurrent processing by multiple warps. However, as we have reviewed in Section II, STMatch let an idle warp to steal half of the stack workloads from another busy warp which causes expensive stack locking/unlocking overheads, while EGSM creates new kernels to handle expensive subtrees which is expensive and adds burden to warp scheduling.

We propose to automatically detect expensive tasks using a timeout mechanism, so that if a task is processed by a warp for longer than a user-specified time threshold τ, it will be decomposed into smaller tasks that are added to a task queue Q_{task}. When a warp becomes idle, it will first attempt to obtain a task from Q_{task} for processing; if Q_{task} is empty, it will then obtain the next chunk of initial tasks for processing. The space of Q_{task} is pre-allocated in the device memory, so subsequent processing does not incur any memory allocation overhead. Fig. 4 summarizes this load balancing strategy of T-DFS.

This strategy keeps the number of tasks small in Q_{task}, since we always prioritize the processing of existing tasks over taking new tasks. As a result, we do not need to set the capacity of Q_{task} to be too large to avoid task overflow. However, since Q_{task} is frequently accessed by all the warps, it is important to design the queue to be lock-free to support high concurrency.

Next, we first describe the design of Q_{task}, and then explain how our timeout mechanism decomposes a straggler task.

Lock-Free Task Queue. Since task decomposition incurs overheads caused by new task creation, insertion to Q_{task}, and fetching from Q_{task}, we try not to create light-workload tasks with small state space subtrees. Following the "StopLevel" concept of STMatch [47], we only create tasks with at most three vertices already matched, i.e., $S = [(u_1, v_{i_1}), (u_2, v_{i_2}), (u_3, v_{i_3})]$, which we denote as $\langle v_{i_1}, v_{i_2}, v_{i_3}\rangle$. Recall that our initial tasks are created from edges (e.g., (v_{i_1}, v_{i_2})), so the only other possible type of task is $S = [(u_1, v_{i_1}), (u_2, v_{i_2})]$, which we store as $\langle v_{i_1}, v_{i_2}, -2\rangle$ where -2 is simply a placeholder for v_{i_3}.

Accordingly, we can organize Q_{task} as an array of N integers (where N is a multiple of 3), used as a ring buffer with a *front* pointer (to the first task) and a *back* pointer (to the next position to append a new task) as shown in Fig. 4. Each task $\langle v_{i_1}, v_{i_2}, v_{i_3}\rangle$ occupies three consecutive integer elements in Q_{task}, and all elements of Q_{task} are initialized as -1, where -1 means the element is empty (i.e., not occupied).

Algorithm 3 shows the algorithms for the initialization, task enqueue, and task dequeue operations of Q_{task}. Specifically, Lines 1–2 initializes an empty queue Q_{task}, where *size* denotes the number of integers in Q_{task} (i.e., the number of tasks is *size*$/3$). The queue is initialized in the GPU global memory at the very beginning of a subgraph-matching job.

Note that the enqueue and dequeue operations are conducted by warps. When we say that a warp enqueues into or dequeues from Q_{task}, the operation is actually conducted by the first thread (with *LANE_ID* = 0) to avoid thread contention within each warp (warp contention still exists when updating Q_{task}).

Lines 3–14 describes how to enqueue a task $\langle v_{i_1}, v_{i_2}, v_{i_3}\rangle$. Specifically, we first examine *size* to check if there is still an empty slot for the task, where we atomically increase size by 3 in Line 4 to register the space usage. Note that since multiple warps may enqueue a task to Q_{task} or dequeue a task from Q_{task} at the same time, atomicity is required when increasing *size*. If the old value of *size* is already N or beyond (Line 4), then Q_{task} is full so we cancel the increment of *size* (Line 5) and return false (Line 6) to signal the failure to enqueue. Otherwise, we forward the *back* pointer for 3 positions to secure the task space in Line 7, where we wrap around by modulo N to treat Q_{task} as a ring buffer. Note that forwarding *back* should still ensure atomicity since multiple warps may enqueue tasks to Q_{task}. Finally, we copy $\langle v_{i_1}, v_{i_2}, v_{i_3}\rangle$ to the 3 secured positions in Lines 7–13 and return *true* to signal success to enqueue in Line 14.

Lines 15–26 describes how to dequeue a task $\langle v_{i_1}, v_{i_2}, v_{i_3}\rangle$. Specifically, we first examine *size* to check if Q_{task} is not empty, where we atomically decrease size by 3 in Line 16 to register the space release. Note that since multiple warps may enqueue a task to Q_{task} or dequeue a task from Q_{task} at the same time, atomicity is required when decreasing *size*.

Algorithm 3 Enqueue and Dequeue Functions of Q_{task}

```
1: $Q_{task} \leftarrow int[N] = \{-1, -1, \ldots, -1\}$
2: $size \leftarrow 0$, $front \leftarrow 0$, $back \leftarrow 0$

3: Function bool enqueue($\langle v_{i_1}, v_{i_2}, v_{i_3} \rangle$)
4:     if atomicAdd(&$size$, 3) $\geq N$ then
5:         atomicSub(&$size$, 3)
6:         return false
7:     $pos \leftarrow$ atomicAdd(&$back$, 3) $\bmod N$
8:     while atomicCAS($Q_{task}[pos]$, $-1$, $v_{i_1}$) $\neq -1$
9:         __nanosleep(10)
10:    while atomicCAS($Q_{task}[pos+1]$, $-1$, $v_{i_2}$) $\neq -1$
11:        __nanosleep(10)
12:    while atomicCAS($Q_{task}[pos+2]$, $-1$, $v_{i_3}$) $\neq -1$
13:        __nanosleep(10)
14:    return true

15: Function bool dequeue($int\& \ v_{i_1}$, $int\& \ v_{i_2}$, $int\& \ v_{i_3}$)
16:    if atomicSub(&$size$, 3) $\leq 0$ then
17:        atomicAdd(&$size$, 3)
18:        return false
19:    $pos \leftarrow$ atomicAdd(&$front$, 3) $\bmod N$
20:    while ($v_{i_1} \leftarrow$ atomicExch($Q_{task}[pos]$, $-1$)) = $-1$
21:        __nanosleep(10)
22:    while ($v_{i_2} \leftarrow$ atomicExch($Q_{task}[pos+1]$, $-1$)) = $-1$
23:        __nanosleep(10)
24:    while ($v_{i_3} \leftarrow$ atomicExch($Q_{task}[pos+2]$, $-1$)) = $-1$
25:        __nanosleep(10)
26:    return true
```

If the old value of *size* is already 0 (Line 16), then Q_{task} is empty so we cancel the decrement of *size* (Line 17) and return false (Line 18) to signal the failure to dequeue. Otherwise, we move *front* backward for 3 positions to release the task space in Line 19, where we wrap around by modulo N to treat Q_{task} as a ring buffer. Finally, we copy elements in the 3 positions to $\langle v_{i_1}, v_{i_2}, v_{i_3} \rangle$ in Lines 19–25 and return *true* to signal success to dequeue in Line 26.

Note that when the queue is full, *front* and *back* point to the same element. So, atomicCAS(.) in Line 8 is required to make sure $Q_{task}[pos]$ has already been cleared to -1 by dequeue(.) before putting v_{i_1} there; and atomicExch(.) at Line 20 is required to ensure that the value to dequeue to v_{i_1} has already been filled by enqueue(.). If the condition does not hold, the thread will sleep for a short time to avoid busy waiting.

Note that our work stealing approach avoids locking stacks (as required by STMatch) that are frequently accessed during DFS-based subgraph matching, as well as the associated busy waiting on some warp that is copying data from a stack for stealing. We only utilize atomic operations highly optimized by CUDA for lightweight contentions on the head and tail of Q_{task}, so the overhead caused by contention is much lower.

The Timeout Mechanism. So far, we have not explained how a task is decomposed when it times out. Fig. 5 illustrates the process of our timeout mechanism. Specifically, suppose we start searching from v_0 which is the root of a state space subtree at time t_0. Each time right before the warp is going to the next level, it first acquires the current time $now()$ to compute the elapsed time of the task $now() - t_0$. If this time is longer than threshold τ, then the current node in the state

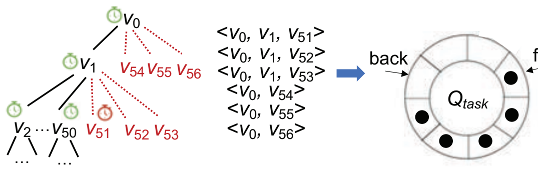

Fig. 5. An Illustration of the Timeout Mechanism

space subtree is wrapped as a task to be added to Q_{task} instead of being processed by the current warp.

For example, in Fig. 5, suppose that $v_0, v_1, \ldots, v_{50}$ are fully traversed within time limit τ, but while attempting to extend deeper from v_{51}, timeout is detected. So instead of continuing extension, the warp adds the extension task $\langle v_0, v_1, v_{51} \rangle$ into Q_{task} and repeats the same operation for each remaining candidate in the same level (note that their $(now() - t_0)$ is still beyond τ since time flows one way). Then it backtracks to the previous level and continues pushing tasks $\langle v_0, v_{54} \rangle$, $\langle v_0, v_{55} \rangle$, $\langle v_0, v_{56} \rangle$ into Q_{task} rather than extending them.

Note that we do not create tasks with more than 3 matched vertices, so if timeout is detected while processing the subtree under v_{50}, the warp needs to continue the backtracking search to finish the entire subtree, and then start task decomposition and enqueueing to Q_{task} after backtracking to v_{51} at level 3.

Algorithm 4 revises Algorithm 2 to include the above timeout mechanism, where the differences and new additions are highlighted in red. Since initial tasks are created from edges of G, *stack*[1] now holds pairs of integer IDs of the form $\langle v_{i_1}, v_{i_2} \rangle$ (i.e., corresponds to the first two levels in state space tree), while the other layers hold individual integer IDs.

Algorithm 4 Stack-Based Warp Backtracking with Timeout

```
1: if $Q_{task}$.dequeue($v_{i_1}, v_{i_2}, v_{i_3}$) = true then
2:     $stack[1] \leftarrow \langle v_{i_1}, v_{i_2} \rangle$, $level \leftarrow 2$
3:     if $v_{i_3} \neq -2$ then   $stack[2] \leftarrow v_{i_3}$, $level \leftarrow 3$
4: else
5:     $stack[1] \leftarrow$ the next edge chunk from $E$, $level \leftarrow 2$
6: $t_0 \leftarrow now()$
7: while $true$ do
8:     if $level < k$ then
9:         if $size[level] = 0$ then
10:            if $level$ = 2 then break
11:            Extend $stack[level]$ along the current path ending at Level $(level - 1)$
12:        $cond \leftarrow level \leq 3$ and $now() - t_0 > \tau$
13:        if $iter[level] < size[level]$ then
14:            if $cond$ = false then
15:                $level \leftarrow level + 1$
16:            else
17:                for each remaining $task$ in $stack[level]$
18:                    if $Q_{task}$.enqueue($task$) = false then
19:                        $t_0 \leftarrow now()$
20:                        goto Line 7
21:                goto Line 23
22:        else
23:            $iter[level] \leftarrow 0$      % Clear $stack[level]$
24:            if $level > 1$ then
25:                $level \leftarrow level - 1$
26:                $iter[level]$++
27:    else
28:        Emit the valid match
29:        $level \leftarrow level - 1$
```

Specifically, Lines 1–3 first attempt to obtain a task from Q_{task} for processing. If Q_{task} is empty, then Lines 4–5 instead fetch a chunk of initial tasks for processing. This strategy minimizes the number of tasks in Q_{task}, and our empirical studies show that a capacity of $N = 3$ million (occupying 12 MB in device memory) is sufficient for all our experiments.

Note that if there is no more edge chunk to fetch in Line 5, the warp exits an outer-loop that repeatedly calls Algorithm 4 to process more chunks. We omit this outer-loop for simplicity.

After the top levels of *stack* are properly populated, and the current level number is set accordingly, Line 6 then records the starting time t_0 before conducting backtracking search with the populated *stack*.

The while-loop from Line 7 then conducts backtracking search with *stack* as in Algorithm 2. However, before moving to the next level in Line 15, we first check the conditions in Line 12 to see if (1) the current level number is within 3 and (2) timeout occurs. If either condition is not met, the warp proceeds to the next level; while otherwise, Lines 16–21 add the remaining tasks in the current level (tracked by *iter*[*level*]) to Q_{task} and go to Line 23 to backtrack to the previous level (to continue adding tasks to Q_{task} due to timeout as t_0 has not changed).

In case Q_{task} becomes full (which is unlikely to happen) as checked by Line 18, then the warp resets t_0 and goes to Line 7 to restore the regular backtracking (*cond* would be *false* in Line 12 to enter Line 15 as t_0 was reset). When the search times out again, task decomposition and addition to Q_{task} will be attempted again (in hope that Q_{task} is no longer full).

Dynamic Stack Space Allocation. Recall from Fig. 3 that the capacity of each level in *stack* is set to the maximum degree in G. However, some real world graphs have very skewed degree distribution and their maximum degree can be very large, up to 8 million [8], [15], [17]. If we use 8 million as the capacity of each level, we need at least hundreds of gigabytes which is way beyond the size of GPU device memory. Moreover, even if the stacks can fit in the device memory, most of the preallocated space will be under-utilized, especially the deeper levels where fewer candidates can be found. So it is desirable to allocate space for the stack levels dynamically on demand, to keep the device memory space occupied by the stacks small (as the remaining space is needed to keep G).

While CUDA's support for dynamic memory management on GPUs is very inefficient, various solutions have been proposed in the last decade [49] which make efficient dynamic space allocation possible. These memory managers take a large preallocate space in the device memory at the beginning, cut the space into smaller blocks of different granularities, and allocate and free block spaces to user programs on demand while taking care of thread contention, fragmentation, etc. In this work, we integrate Ouroboros [48] as our dynamic memory manager which allows threads to call 'malloc' and 'free' to request/release memory space of various granularities ranging from 16 B to 8 KB.

Fig. 6 illustrates our dynamic stack space management. Specifically, we divide the preallocated space for dynamic

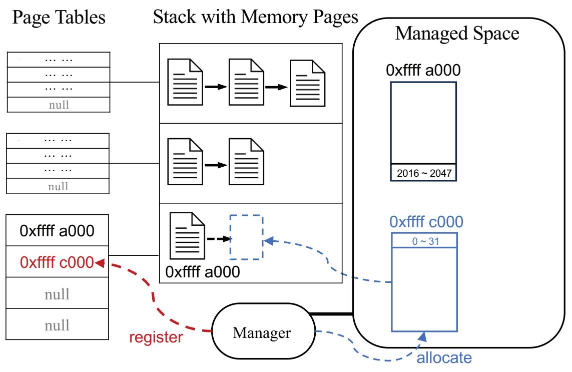

Fig. 6. Page Tables and Stack Memory Management

memory allocation into pages of the same size (8 KB by default), which is managed by Ouroboros. Each level of a stack is regarded (logically) as a list of pages as shown in the middle of Fig. 6. If the current pages in a level are fully occupied, a new page can be requested from Ouroboros to be appended to the list. Since the pages are actually in the preallocated space managed by Ouroboros, each stack level is actually stored as a *page table* as shown on the left of Fig. 6. Here, a page table is a small address array with a fixed size (40 addresses by default), where elements are initialized as *null* and will be replaced by used page addresses later as needed. Note that each stack level can keep up to 40×8 KB = 320 KB data, or equivalently, 81,920 integer vertex IDs of 32 bits each, which we find to be sufficiently large for all our datasets and experiments since a stack level is used to store vertex candidates that are the result of set intersections of neighbor lists (c.f., Eq (1)), tending to be much smaller than the maximum degree of G). Note that even if we want to allow up to 8 million vertex IDs in a stack level, we only need to adjust the page table size as 4000 which is still well affordable since the total space used by page tables is only $4000 \cdot k \cdot n_{warp}$, where $k = |V_Q|$ is the number of levels in a stack, and n_{warp} is the number of warps in the kernel.

Releasing used pages is possible. For example, assume we have n pages in a stack level, then we expand new candidates into this level, if it uses no more than $n/4$ pages, then we can free the last $n/2$ pages. However, we find this to be not necessary in our experiments since the memory space occupied by all the pages is very small even without page releasing.

Recall that in Line 11 of Algorithm 4, the 32 threads of the current warp need to compute the candidates of a vertex at position *iter*[*level* − 1] of *stack*[*level* − 1] (let the candidate number be n_{cand}), and place them to the first n_{cand} consecutive positions in *stack*[*level*]. Since *stack*[*level*] is now paged, we need to overload the [] operator for the threads of a warp to access a specified position(s) *pos* in the stack level *stack*[*level*]. The algorithm is shown in Algorithm 5.

Specifically, we first obtain *PAGE_ID* at Line 1 which is the page index of *stack*[*level*] to access, and obtain *OFFSET* at Line 2 which is the position to access in that page. Since this page may not exist yet, we obtain a mask in Line 4 which is a 32-bit bitmap where the i^{th} bit is 1 if the i^{th} thread enters the if-branch from Line 3, and 0 otherwise. Only those active

Algorithm 5 Paged Element Access in a Stack Level

```
Operator Overloading: stack[level][pos]
Let PAGE_TABLE be the page table of stack[level]
1: PAGE_ID ← pos / PAGE_SIZE
2: OFFSET ← pos mod PAGE_SIZE
3: if PAGE_TABLE[PAGE_ID] = null then
4:     mask ← __activemask()
5:     leader ← __ffs(mask) − 1
6:     if LANE_ID = leader then
7:         Request a new page NP
8:         PAGE_TABLE[PAGE_ID] ← NP
9:     __syncwarp(mask)
10: return PAGE_TABLE[PAGE_ID][OFFSET]
```

threads execute the if-branch, where in Line 5, a unique leader thread is determined using the position of the least significant bit set to 1 in *mask*. Only this leader thread will request a new page from the memory manager at Line 7 and record its address in the *PAGE_TABLE* at Line 8. The other active threads will wait for the new page at Line 9. Once the leader is done, all the active threads can then access the new page *PAGE_TABLE*[*PAGE_ID*]. On the other hand, those threads that does not enter the if-branch can directly access the correct page at Line 10 without delay.

To illustrate, the right of Fig. 6 shows how a warp accesses the next 32 elements (i.e., vertex IDs) from *stack*[*level*] for writing when they cross the page boundary. Note that candidates are written by the 32 threads of a warp to *stack*[*level*] in batches of at most 32 elements each, so each batch of elements can cross at most one page boundary (as a page has 8KB). A batch may write less that 32 elements since when computing candidates by set intersection $A \cap B$, each thread checks an element $a \in A$ with binary search against B, and if binary search fails, $a \notin A \cap B$. After the 32 elements in A are checked by a warp with binary search, valid ones will be compacted using the warp-level compacting operation illustrated in Fig. 8 of [19] (using ballot scan) to be placed into consecutive elements in *stack*[*level*]. In Fig. 6, suppose that the 32 threads of a warps need to write to the positions 2032–2063 in *stack*[*level*] (i.e., all the 32 elements are valid candidates). According to Algorithm 5, threads 1–16 write positions 2032–2047 of *stack*[*level*] located at the end of page '0xffffa000', and threads 17–32 write positions 2048–2063 of *stack*[*level*] located at the beginning of a new page '0xffffc000' requested by the leader thread 17.

Our dynamic page allocation scheme saves a lot of memory compared with using an array of d_{max} elements for every stack level. As Fig. 6 illustrates, deeper levels tend to maintain fewer candidates, so most space would be wasted in deeper levels if we use d_{max} as the capacity. In Section IV-G, we find that our page-based design saves up to 86% of memory compared to the array-based solution, meaning that at least 86% of the memory in the array-based solution is wasted.

Algorithm Optimizations. So far, our proposed techniques are general for DFS-based subgraph search on GPUs. Next, we present two algorithmic optimizations specific to subgraph matching: 1) edge filtering and 2) set intersection result reuse.

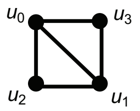

Fig. 7. An Example to Illustrate Set Intersection Result Reuse

These techniques have been shown to effectively speed up subgraph matching in existing GPU systems [29]–[31], [40], and our experiments reported in our online appendix [10]) also confirm their effectiveness in our T-DFS framework.

Edge filtering happens in Line 5 of Algorithm 4 when a warp reads the next chunk of initial tasks (i.e., edges) into *stack*[1]. Specifically, for each edge (v_{i_1}, v_{i_2}) to match (u_1, u_2), if any of the following four conditions does not hold, the edge (i.e., task) is filtered: (1) $degree(v_{i_1}) \geq degree(u_1)$; (2) $degree(v_{i_2}) \geq degree(u_2)$; (3) $label(v_{i_2}) = label(u_2)$; and (4) $label(v_{i_2}) = label(u_2)$. Additionally, we also filter candidates based on their labels during subgraph extension in Line 11 of Algorithm 4.

We next illustrate the idea of set intersection result reuse using the query graph G_Q shown in Fig. 7. Suppose we match in the order $\pi = [u_0, u_1, u_2, u_3]$, then the candidates of u_2 is obtained through $N(v_0) \cap N(v_1)$ where v_0 and v_1 are the vertices matched to u_0 and u_1, respectively. Similarly, the candidates of u_3 is obtained through $N(v_0) \cap N(v_1)$ since u_3 only connects with u_0 and u_1. Apparently, the results of $N(v_0) \cap N(v_1)$ which is the candidates of u_2 can be reused when getting the candidates of u_3 since $B^\pi(u_2) \subseteq B^\pi(u_3)$ where $B^\pi(u)$ refers to the backward neighbors of u in Q.

In general, we can determine whether computing the candidates of u_j can reuse the candidates of u_i $(i < j)$ by checking if $BN(u_i) \subseteq BN(u_j)$; if so, then we can compute the candidates of u_j directly from the candidates of u_i.

To integrate this approach in Algorithm 4, we first preprocess G_Q to derive the dependencies of backward neighbors between vertices on the CPU, the cost of which is negligible as G_Q is small. By the time when we need to compute the candidates of u_j which rely on those of u_i, we can directly use the results in *stack*[i] to continue processing with the remaining neighbors. For example, suppose $B^\pi(u_i) = \{u_0, u_1\}$ and $B^\pi(u_j) = \{u_0, u_1, u_2\}$, so $B^\pi(u_i) \subseteq B^\pi(u_j)$. If this technique is not enabled, *stack*[j] $= N(v_0) \cap N(v_1) \cap N(v_2)$; while if this approach is enabled, we can compute *stack*[j] = *stack*[i] $\cap N(v_2)$ rather than computing from scratch.

IV. Experiments

This section reports our experiments to verify the efficiency of T-DFS including each of its techniques, and to compare it with the other two DFS solutions for subgraph matching on GPUs, namely, STMatch [47] and EGSM [43], which were shown to outperform the other previous GPU solutions for subgraph matching such as [50], [54]. We also include PBE [29] as the typical BFS solution on GPUs.

All of our experiments were conducted on a node of the Polaris supercomputer in Argonne National Laboratory, equipped with 4 Nvidia A100 GPUs, each with 40 GB device memory. We used nvcc-11.6.0 to compile all the code. For the

TABLE I
DATASETS

Graph	$\|V\|$	$\|E\|$	Avg deg	Max deg
Amazon	334,863	925,782	5.5	549
DBLP	317,080	1,049,866	6.6	343
YouTube	1,134,890	2,987,624	5.3	28,754
web-Google	875,713	4,322,051	9.9	6332
imdb-2021	1,224,268	5,369,400	8.8	833
cit-Patents	3,774,768	16,518,947	8.8	793
Pokec	1,632,803	22,301,964	27.3	14,854
soc-facebook	3,097,165	23,667,394	15.3	4915
Orkut	3,702,441	117,185,083	76.3	33,313
soc-sinaweibo	58,655,849	261,321,033	8.9	278,489
Datagen-90-fb	12,857,671	1,049,527,225	163.3	4207
Friendster	65,608,366	1,806,067,135	55.1	5214

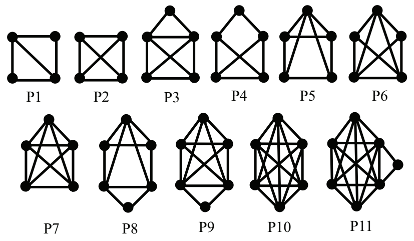

Fig. 8. Query Patterns for Evaluation

timeout mechanism of T-DFS, we set the user-specified time threshold τ as 10 ms by default. Our source code is released at https://github.com/lyuheng/tdfs.

A. Datasets

We used 12 real-world graph datasets as shown in Table I in our experiments. Among them, Amazon [1], DBLP [5], YouTube [18], imdb-2021 [9], soc-facebook [16], Orkut [11], Friendster [6], Pokec [13] and soc-sinaweibo [14] are social networks, web-Google [7] is a web graph, cit-Patents [12] is a citation network, and Datagen-90-fb [4] is a synthetic dataset generated using the LDBC benchmark [22].

We classify the data graphs G into two categories based on $|E|$: the first 8 are moderate-sized graphs, and the last 4 are big graphs. The moderate-sized graphs are unlabeled, but to allow our experiments to finish in reasonable amount of time on the 4 big graphs, we increase the selectivity of G_Q by randomly assigning 4 labels to the data vertices in G.

As for the query graph G_Q, we consider the 11 patterns shown in Fig. 8, which are commonly used in existing works [29], [35] for performance evaluation. When considering labeled patterns, we let all the query vertices in P1–P11 take the same label, and add 11 more patterns P12–P22 which have the same structure as P1–P11 but each vertex $u_i \in G_Q$ takes the label $(i \mod 4)$.

B. Comparing T-DFS with Existing GPU Solutions

We compare our T-DFS framework with three existing works, namely STMatch [47], EGSM [43] and PBE [29]. STMatch and EGSM both adopt a DFS approach, support labeled and unlabeled subgraph matching, and are shown to be faster than previous works [50], [54]. PBE is a BFS approach with pipelined memory management to prevent device memory overflow, but only supports unlabeled subgraph matching.

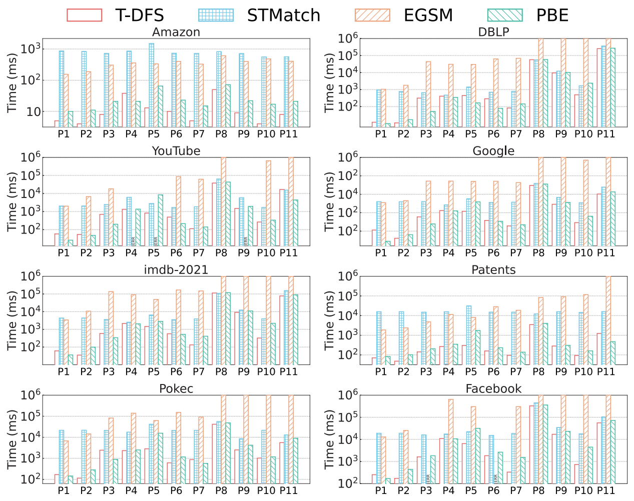

Fig. 9. Comparison of T-DFS, STMatch, EGSM & PBE on Unlabeled Graphs

In the released source code of STMatch [2], we found that it assumed every query node to have a different label, so we modified their code to accommodate the general case where two query nodes may have the same label. The default capacity STMatch for each stack level is 4096 vertex IDs, but since some of our graphs have a much higher d_{max}, the default setting may leads to wrong results, so we set the capacity to d_{max} instead unless otherwise stated.

Fig. 9 shows the running time of T-DFS, STMatch, EGSM and PBE for subgraph matching on the 8 unlabeled graphs. Our T-DFS clearly outperforms DFS solutions STMatch and EGSM, with an average speedup of about 42.0× and 360.1×, respectively. Compared with PBE, T-DFS is better in majority of the cases, with an average speedup of about 2.0×.

STMatch is slower than T-DFS, which is due to a poor implementation choice. Specifically, recall from Algorithm 1 that before executing Line 5, we also need to make sure v is not already matched to another query vertex in S (i.e., such a vertex is removed from the candidates). T-DFS conducts set intersections and vertex removal together which is lightweight, while STMatch treats vertex removal as an independent set-difference operation which leads to more rounds of set operations to compute the candidate set.

EGSM is the slowest since it does not conduct automorphism check as STMatch and T-DFS do, which leads to a lot of redundant computations in the unlabeled setting. T-DFS integrates the BLISS [3] library for computing the automorphism groups of the input queries, which are then used to generate constraints between vertices to break the pattern symmetry, hence avoiding redundant computation. On YouTube and Facebook, EGSM failed when matching some patterns likely due to implementation issues, so we mark their results as 'ERR'.

The performance of PBE is the fastest among 3 baselines. Compared with T-DFS, PBE is generally slower but is closer to T-DFS when degree distribution is more biased (as measured by d_{max} shown in Table I) since PBE's BFS approach does not suffer from imbalanced workloads as much as our warp-based

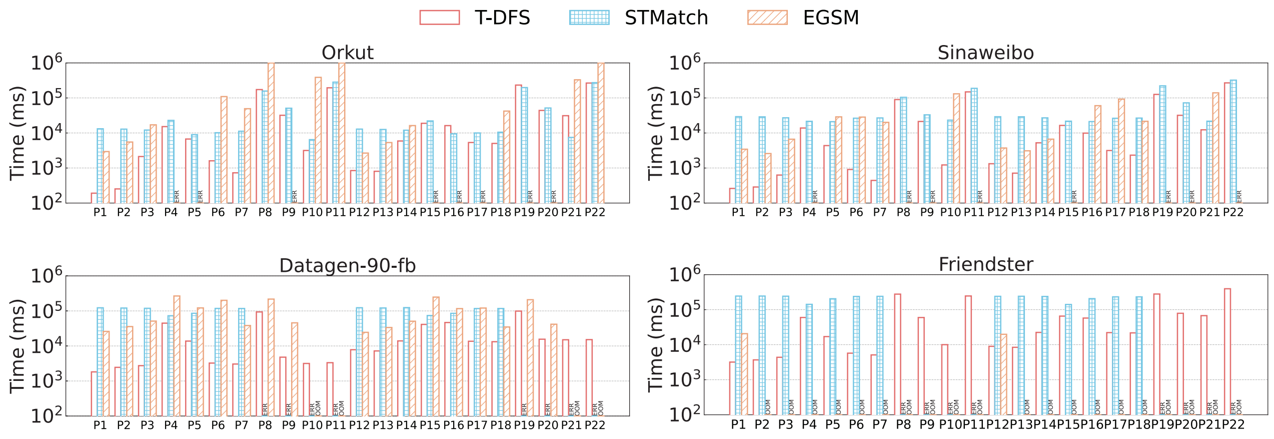

Fig. 10. Comparison of T-DFS, STMatch & EGSM on Big Graphs

DFS solution. Specifically, on YouTube and Pokec where d_{max} is large, T-DFS only achieves an average improvement of 1.6× speedup (and even slower than PBE in a few patterns), while on the other graphs where d_{max} is smaller, T-DFS achieves a better average improvement of 2.2× speedup.

Fig. 10 shows the running time of T-DFS, STMatch and EGSM for subgraph matching on the 4 big labeled graphs. PBE does not support labeled query graphs so it is not included. We can see that T-DFS still outperforms STMatch and EGSM by an average speedup of 20× and 15×, respectively.

Moreover, T-DFS is much faster on P1–P11 than on P12–P22 since vertices in P1–P11 have the same label, so set intersection result reuse is more effective.

Unlike T-DFS and EGSM that conduct initial filtering of unpromising edges (i.e., initial tasks) using GPU warps in parallel, STMatch implements edge filtering as a preprocessing step on the host by a single CPU core before invoking a kernel for subgraph matching. We find that while the preprocessing step is negligible on small graphs, it can become a bottleneck on big graphs. For example, it takes around 80 seconds for all the tested query patterns on the largest graph 'Friendster', which accounts for up to 58% of the total time. Even when compared with only STMatch's subgraph matching time, T-DFS still wins in most cases because STMatch treats vertex removal as an independent set-difference operation which leads to more rounds of set operations to compute the candidate set.

EGSM is slower than T-DFS, and even worse, it failed to complete in some experiments. For example, on Friendster, EGSM reports an 'Out of Memory' (OOM) error for most patterns due to the high space demand for building the Cuckoo Trie index (CT-index), which maintains compact candidates (which can be many) after advanced pruning. EGSM finishes for P1 and P12 on Friendster since they only have 5 edges, so the space used by CT-index for maintaining edge candidates is within the device memory space limit. Moreover, even in those cases where EGSM does not fail, EGSM is still slower than T-DFS because EGSM relies on the CT-index to find the neighbors to extend, but to access each neighbor in the CT-index, we need three times of memory accesses since the

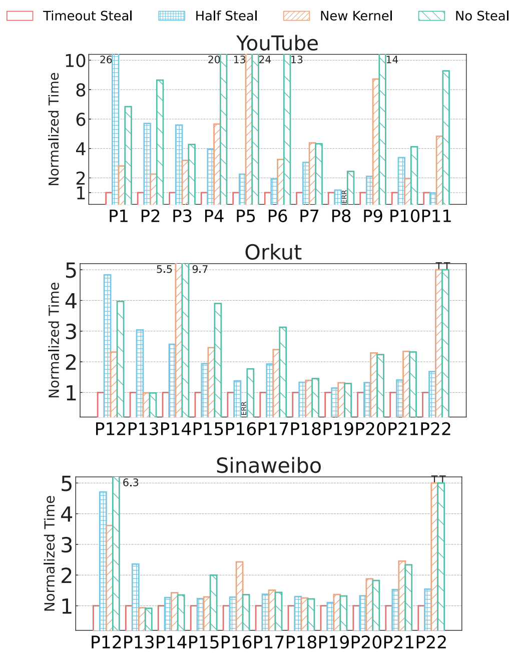

Fig. 11. Comparison of Work Stealing Strategies ('T' means > 1000 s)

CT-index consists of 3 levels, *cuc*, *off* and *nbr* (see Fig. 3 of [43]). If the preprocessing of advanced candidate pruning in CT-index cannot compensate for the extra memory access cost, reading from CT-index is more expensive than simply using the CSR format. EGSM also failed on the other three graphs for some queries due to illegal memory access, which is likely caused by the bugs in its implementation.

C. Comparison of Load Balance Strategies

To compare our proposed work stealing strategy that is based on timeout mechanism (denoted by 'Timeout Steal') with that of STMatch as illustrated by Fig. 2 (denoted by 'Half Steal') and the 'new kernel' strategy of EGSM reviewed at the end of Section II (denoted by 'New Kernel'), we also implemented Half Steal and New Kernel in our T-DFS framework. Fig. 11 compares the performance of 'Timeout

TABLE II
ABLATION STUDY OF τ ON YOUTUBE (UNIT: MILLISECOND)

τ	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
1	105	74	711	1435	1134	610	124	**37,950**	1493	**139**	20,927
10	**58**	**54**	**702**	**1334**	**825**	**493**	**113**	37,977	**1492**	140	**16,666**
100	521	148	919	1499	959	707	252	38,349	1659	428	25,113
1000	386	438	9049	5439	4788	2406	454	41,172	5015	1077	84,111
∞	408	438	2946	27,768	19,881	6962	457	93,559	22,757	1101	155,720

TABLE III
ABLATION STUDY OF τ ON POKEC (UNIT: MILLISECOND)

τ	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
1	170	118	3374	4246	3880	1496	**811**	42,263	2801	1048	5629
10	**169**	**115**	**2434**	**1321**	2122	**612**	866	**41,568**	**2522**	**1025**	**5571**
100	312	262	2456	2372	**2114**	3057	905	42,623	2859	1413	5680
1000	833	1164	15,489	2479	2160	3314	1279	45,193	4147	1347	8676
∞	833	1165	5874	82,698	2161	7528	1843	47,342	4913	2006	37,049

Steal', 'Half Steal', 'New Kernel' and 'No Steal' (i.e., when work stealing is disabled). Due to space limit, we only show the results on YouTube, Orkut and Sinaweibo; the results on the other graphs are similar. We can see that our 'Timeout Steal' strategy clearly outperforms 'Half Steal' since there is no costly locking overhead. We notice that sometimes, this negative impact of locking overheads is so severe that 'Half Steal' becomes even slower than 'No Steal'. For example, on Orkut, 'Half Steal' is 21% (resp. 2.08×) slower than 'No Steal' on P12 (resp. P13), respectively. Note that 'Half Steal' still speeds up computation in most cases as compared with 'No Steal', but even in these experiments, 'Timeout Steal' is still the clear winner thanks to its lightweight and lock-free design.

The 'New Kernel' strategy is also clearly beaten by our 'Timeout Steal' strategy since 'New Kernel' requires extra memory allocations to create new stacks before a new kernel starts. This not only leads much more memory consumption, but also causes program failure for P8 on YouTube and P16 on Orkut somehow (we are not clear about the reason, but they also fail if we run EGSM directly for the same reason of creating new kernels, showing that 'New Kernel' may not be robust). Besides, 'New Kernel' requires users to manually tune the fanout threshold for invoking a new kernel to find a good one, which is highly sensitive to the input data and query graph. In contrast, our default timeout threshold in 'Timeout Steal' consistently achieves the best performance regardless of input graphs.

D. Effect of the Timeout Threshold

We also conducted experiments to find the best timeout threshold τ for 'Timeout Steal', and our finding is that the default $\tau = 10$ ms works consistently the best or nearly the best in various datasets. Without loss of generality, Table II shows the running time of matching P1–P11 on YouTube when we vary τ as 1, 10, 100, 1000 and ∞. We can see that $\tau = 10$ ms gives the best (or very close to the best) performance on YouTube for all the matching patterns, and the running time for $\tau = 1$ ms and $\tau = 100$ ms are clearly longer than when $\tau = 10$ ms. Table III shows the results on Pokec, where we can obtain similar observations.

Note that when τ is set too large, stragglers cannot be decomposed and are only computed by a few warps while the other warps are just idle, so the running time can become very long. For example, for P4 on Pokec, 'Timeout Steal' with $\tau = 10$ ms achieves 62.6× speedup compared to 'No Steal' (i.e. $\tau = \infty$). On the other hand, if we decrease τ too aggressively and set it to 1 ms, the performance will not be improved, since excessive task decomposition will incur more overheads of task creation and management than if in-place execution is used.

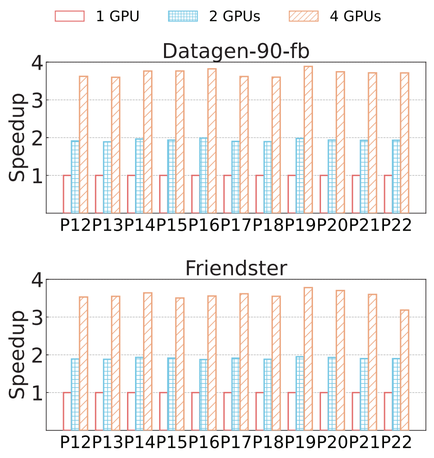

Fig. 12. Scale-up Performance on Multiple GPUs

E. Scale-up Performance on Multiple GPUs

Recall that T-DFS can scale to multiple GPUs by partitioning the initial edges (i.e., initial tasks) in a round-robin manner where the i^{th} edge is assigned to the $(i \bmod \text{NUM\_GPU})^{\text{th}}$ GPU. We tested the scale-up performance of T-DFS and found that T-DFS can achieve ideal speedup.

Without loss of generality, Fig. 12 shows the multi-GPU speedup of T-DFS on the 2 largest graphs, Datagen-fb-90 and Friendster. We can see that all the tested queries can achieve a speedup proportional to number of GPUs.

F. Effect of Increasing Label Selectivity

In order to investigate whether T-DFS is still more favorable than the baselines when the labels in data graph are more selective, we gradually increase the number of labels $|L|$ of the largest graph Friendster from 4 to 16, and then evaluate using some 6-node patterns, P8–P10, as shown in Table IV. STMatch is not compared since we observe errors during its execution. Table IV shows that T-DFS still outperforms EGSM even though EGSM exploits CT-index for fast candidate pruning and lookup. This is because CT-index requires one extra memory access which cannot be compensated by its pruning power. However, we see that CT-index is more preferable when $|L|$ increases, so there is a potential for EGSM to finally surpass T-DFS when $|L|$ is very large, but in those cases the

TABLE IV
EFFECT OF INCREASING LABEL SELECTIVITY ON FRIENDSTER

Label Size $\vert L\vert$	P8		P9		P10	
	Ours	EGSM	Ours	EGSM	Ours	EGSM
4	**277,051**	OOM	**59,650**	OOM	**10,022**	OOM
8	**11,434**	15,364	**2747**	9607	**1313**	14,078
12	**2263**	2999	**886**	2277	**705**	2869
16	**924**	977	**603**	971	**558**	1077

TABLE V
STACK MEMORY CONSUMPTION ON POKEC (UNIT: GB)

Method	P1	P2	P3	P4	P5	P6	P7
Page-based	0.184	0.184	0.184	0.223	0.213	0.201	0.184
Array-based	1.437						

TABLE VI
EXECUTION TIME ON POKEC (UNIT: SECOND)

Method	P1	P2	P3	P4	P5	P6	P7
Page-based	0.87	0.58	7.65	7.25	11.42	4.33	1.49
Array-based	0.16	0.11	2.43	2.32	2.83	0.61	0.86
STMatch	19.60	19.64	19.13	15.56	36.96	19.17	19.38

job time itself is already below 1 second so further speedup by parallelization is not essential.

G. Effect of Dynamic Stack Space Allocation

Recall that our dynamic page allocation scheme saves a lot of memory compared with using an array of d_{max} elements (i.e., vertex IDs) for every stack level. We have conducted experiments to verify the effectiveness of our page-based solution in saving device memory and ensuring result correctness as compared with the array-based baseline solution.

To ensure that the intermediate candidates do not overflow the buffer (for a stack level) causing inaccurate results, we have to allocate a buffer capacity of d_{max} elements which can be huge in order to guarantee result correctness. However, real graphs are often power-law graphs that can have very skewed degree distribution and hence a large d_{max}, so the total space consumed by the stacks can be huge.

Due to the limited device memory space of GPUs, STMatch only allocates a capacity of 4096 elements for each stack level. Although this is empirically sufficient for many graphs with balanced degree distribution, we observed result incorrectness on Youtube, Pokec, Orkut and Sinaweibo, which are graphs with very large d_{max}. For example, if we match P3 on Pokec and use STMatch's default capacity of 4096 elements, the results are incorrect since STMatch finds 2 million more matchings than the correct number of matchings.

Also note that even if d_{max} is not too large, allocating every stack level with a capacity of d_{max} elements causes huge memory under-utilization in deeper levels.

Our page-based solution as illustrated in Fig. 6 dynamically allocates memory space for each stack level based on its demand. We next show that it significantly improves the memory usage and meanwhile, guarantees the result correctness.

Table V and VII compares the stack memory consumption of our page-based solution v.s. array-based baseline on Pokec and YouTube, respectively. We can see that the page-based design saves around 86% of memory on Pokec and 93% of memory on Youtube, compared to array-based solution, therefore confirming that it is wasteful to allocate array with a capacity of d_{max} elements to every stack level.

However, the running time of our page-based solution is slower than the array-based baseline (provided with sufficient stack-level capacity to ensure result correctness), as is clear from Table VI and VIII where we compare their running

TABLE VII
STACK MEMORY CONSUMPTION ON YOUTUBE (UNIT: GB)

Method	P1	P2	P3	P4	P5	P6	P7
Page-based	0.184	0.184	0.184	0.209	0.204	0.185	0.184
Array-based	2.782						

TABLE VIII
EXECUTION TIME ON YOUTUBE (UNIT: SECOND)

Method	P1	P2	P3	P4	P5	P6	P7
Page-based	0.21	0.08	2.22	3.13	3.07	0.78	0.22
Array-based	0.06	0.05	0.70	1.33	0.82	0.49	0.11
STMatch	1.89	1.89	2.33	6.03	2.52	1.56	1.72

time together with that of STMatch. This is within expectation since GPU favors coalesced memory access, but the paged memory access incurs condition-checking on page existence and cross-page access (see Line 3 in Algorithm 5) which lead to longer IO latency. However, compared with STMatch, our page-based solution is still significantly faster while greatly reducing unnecessary memory consumption and guaranteeing result correctness.

V. CONCLUSIONS AND FUTURE WORK

This paper proposed a novel approach, called T-DFS, to address the existing problems of executing depth-first search on GPUs, and used subgraph matching as an example to prove its effectiveness. By decomposing straggler tasks into smaller tasks through a timeout mechanism, the workload is evenly divided to all warps. Those tasks are managed by a lock-free circular queue in the memory which distributes tasks to idle warps concurrently. A flexible and fine-grained dynamic memory allocation for stack spaces was proposed based on memory paging, to avoid the stack space allocation pitfalls of existing works. Moreover, effective optimizations are integrated including edge filtering and set intersection result reuse. Extensive experiments showed that T-DFS outperforms existing works by orders of magnitude.

As a future work, we plan to explore using BFS subgraph extension initially when the extended subgraphs fit in the device memory, and switch to DFS processing when the next level of subgraphs cannot fit in device memory. This hybrid BFS-DFS solution is promising since ESGM has shown that BFS is more efficient than DFS due to more coalesced memory access patterns, but we need to divide the device memory between (i) subgraph buffers for BFS extension and (ii) stacks for DFS extension, and we will explore efficient memory allocation strategies. It is also interesting to integrate the CT-index approach of EGSM for candidate pruning as a preprocessing step, and explore when this would be more beneficial than the current T-DFS in terms of query selectivity, especially when there are more vertices in a query graph.

Acknowledgment. This work was supported by DOE ECRP Award 0000274975, NSF OIA-2229394, South Big Data Innovation Hub 2022 S.E.E.D.S. Award, NSF OAC-2414474, NSF OAC-2414185, NSF OAC-2313191 and NSF-2152085.

References

[1] Amazon. https://snap.stanford.edu/data/com-Amazon.html.

[2] An STMatch Version. https://github.com/HPC-Research-Lab/STMatch/commit/f98462.

[3] BLISS. http://www.tcs.hut.fi/Software/bliss/.

[4] datagen-90-fb. https://ldbcouncil.org/benchmarks/graphalytics/.

[5] DBLP. https://snap.stanford.edu/data/com-DBLP.html.

[6] Friendster. https://snap.stanford.edu/data/com-Friendster.html.

[7] Google. https://snap.stanford.edu/data/web-Google.html.

[8] gsh-2005. https://law.di.unimi.it/webdata/gsh-2015/.

[9] imdb-2021. https://law.di.unimi.it/webdata/imdb-2021/.

[10] Online Appendix. https://github.com/lyuheng/tdfs/blob/main/appendix.pdf.

[11] Orkut. https://snap.stanford.edu/data/com-Orkut.html.

[12] Patents. https://snap.stanford.edu/data/cit-Patents.html.

[13] Pokec. https://snap.stanford.edu/data/soc-Pokec.html.

[14] Sinaweibo. https://networkrepository.com/soc-sinaweibo.php.

[15] sk-2005. https://law.di.unimi.it/webdata/sk-2005/.

[16] socfb-A-anon. https://networkrepository.com/socfb-A-anon.php.

[17] uk-2007. https://law.di.unimi.it/webdata/uk-2007-04/.

[18] YouTube. https://snap.stanford.edu/data/com-Youtube.html.

[19] A. Ahmad, L. Yuan, D. Yan, G. Guo, J. Chen, and C. Zhang. Accelerating k-core decomposition by a GPU. In *ICDE*, pages 1818–1831. IEEE, 2023.

[20] M. Almasri, Y. Chang, I. E. Hajj, R. Nagi, J. Xiong, and W. W. Hwu. Parallelizing maximal clique enumeration on gpus. *CoRR*, abs/2212.01473, 2022.

[21] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W. Hwu. Parallel k-clique counting on gpus. In *ICS*, pages 21:1–21:14. ACM, 2022.

[22] P. A. Boncz, I. Fundulaki, A. Gubichev, J. L. Larriba-Pey, and T. Neumann. The linked data benchmark council project. *Datenbank-Spektrum*, 13(2):121–129, 2013.

[23] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[24] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *EuroSys*, pages 32:1–32:12. ACM, 2018.

[25] X. Chen, R. Dathathri, G. Gill, and K. Pingali. Pangolin: An efficient and flexible graph mining system on CPU and GPU. *Proc. VLDB Endow.*, 13(8):1190–1205, 2020.

[26] V. V. dos Santos Dias, C. H. C. Teixeira, D. O. Guedes, W. M. Jr., and S. Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *SIGMOD*, pages 1357–1374. ACM, 2019.

[27] G. Guo, D. Yan, M. T. Özsu, Z. Jiang, and J. Khalil. Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach. *Proc. VLDB Endow.*, 14(4):573–585, 2020.

[28] G. Guo, D. Yan, L. Yuan, J. Khalil, C. Long, Z. Jiang, and Y. Zhou. Maximal directed quasi -clique mining. In *ICDE*, pages 1900–1913. IEEE, 2022.

[29] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K. Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *SIGMOD*, pages 1067–1082. ACM, 2020.

[30] W. Guo, Y. Li, and K. Tan. Exploiting reuse for GPU subgraph enumeration. *IEEE Trans. Knowl. Data Eng.*, 34(9):4231–4244, 2022.

[31] W. Guo, Y. Li, and K. Tan. Exploiting reuse for GPU subgraph enumeration (extended abstract). In *ICDE*, pages 3765–3766. IEEE, 2023.

[32] S. Jain and C. Seshadhri. The power of pivoting for exact clique counting. In *WSDM*, pages 268–276. ACM, 2020.

[33] K. Jamshidi, R. Mahadasa, and K. Vora. Peregrine: a pattern-aware graph mining system. In *EuroSys*, pages 13:1–13:16. ACM, 2020.

[34] J. Jenkins, I. Arkatkar, J. D. Owens, A. N. Choudhary, and N. F. Samatova. Lessons learned from exploring the backtracking paradigm on the GPU. In *Euro-Par*, volume 6853 of *Lecture Notes in Computer Science*, pages 425–437. Springer, 2011.

[35] G. Jiang, C. Q. Zhou, T. Jin, B. Li, Y. Zhao, Y. Li, and J. Cheng. Vsgm: View-based gpu-accelerated subgraph matching on large graphs. In *SC*, pages 739–753. IEEE Computer Society, 2022.

[36] J. Khalil, D. Yan, G. Guo, and L. Yuan. Parallel mining of large maximal quasi-cliques. *VLDB J.*, 31(4):649–674, 2022.

[37] G. Liu and L. Wong. Effective pruning techniques for mining quasi-cliques. In *PKDD*, volume 5212 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2008.

[38] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu. Graphzero: A high-performance subgraph matching system. *ACM SIGOPS Oper. Syst. Rev.*, 55(1):21–37, 2021.

[39] D. Mawhirter and B. Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *SOSP*, pages 509–523. ACM, 2019.

[40] T. Shi, M. Zhai, Y. Xu, and J. Zhai. Graphpi: high performance graph pattern matching through effective redundancy elimination. In *SC*, page 100. IEEE/ACM, 2020.

[41] S. Sun and Q. Luo. Parallelizing recursive backtracking based subgraph matching on a single machine. In *ICPADS*, pages 42–50. IEEE, 2018.

[42] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *SIGMOD*, pages 1083–1098. ACM, 2020.

[43] X. Sun and Q. Luo. Efficient gpu-accelerated subgraph matching. *Proc. ACM Manag. Data*, 1(2):181:1–181:26, 2023.

[44] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440. ACM, 2015.

[45] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[46] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *OSDI*, pages 763–782. USENIX Association, 2018.

[47] Y. Wei and P. Jiang. Stmatch: Accelerating graph pattern matching on GPU with stack-based loop optimizations. In *SC*, pages 53:1–53:13. IEEE, 2022.

[48] M. Winter, D. Mlakar, M. Parger, and M. Steinberger. *Ouroboros*: virtualized queues for dynamic memory management on gpus. In *ICS '20: 2020 International Conference on Supercomputing, Barcelona Spain, June, 2020*, pages 38:1–38:12. ACM, 2020.

[49] M. Winter, M. Parger, D. Mlakar, and M. Steinberger. Are dynamic memory managers on gpus slow?: a survey and benchmarks. In *PPoPP*, pages 219–233. ACM, 2021.

[50] L. Xiang, A. Khan, E. Serra, M. Halappanavar, and A. Sukumaran-Rajam. cuts: scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure. In *SC*, page 69. ACM, 2021.

[51] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W. Ku, and J. C. S. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *ICDE 2020*, pages 1369–1380. IEEE, 2020.

[52] D. Yan, G. Guo, J. Khalil, M. T. Özsu, W. Ku, and J. C. S. Lui. G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing. *VLDB J.*, 31(2):287–320, 2022.

[53] L. Zeng, L. Zou, and M. T. Özsu. SGSI - A scalable gpu-friendly subgraph isomorphism algorithm. *IEEE Trans. Knowl. Data Eng.*, 35(11):11899–11916, 2023.

[54] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang. GSI: gpu-friendly subgraph isomorphism. In *ICDE*, pages 1249–1260. IEEE, 2020.

[55] Y. Zhou, J. Xu, Z. Guo, M. Xiao, and Y. Jin. Enumerating maximal *k*-plexes with worst-case time guarantee. In *AAAI*, pages 2442–2449. AAAI Press, 2020.