9-2022

Exploiting reuse for GPU subgraph enumeration

Wentiao GUO
National University of Singapore

Yuchen LI
Singapore Management University, yuchenli@smu.edu.sg

Kian-Lee TAN
National University of Singapore

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3035564, IEEE Transactions on Knowledge and Data Engineering

Exploiting Reuse for GPU Subgraph Enumeration

Wentian Guo, Yuchen Li, *Member, IEEE,* Kian-Lee Tan, *Senior Member, IEEE*

Abstract—Subgraph enumeration is important for many applications such as network motif discovery, community detection, and frequent subgraph mining. To accelerate the execution, recent works utilize graphics processing units (GPUs) to parallelize subgraph enumeration. The performances of these parallel schemes are dominated by the set intersection operations which account for up to 95% of the total processing time. (Un)surprisingly, a significant portion (as high as 99%) of these operations is actually redundant, i.e., the same set of vertices is repeatedly encountered and evaluated. Therefore, in this paper, we seek to salvage and recycle the results of such operations to avoid repeated computation. Our solution consists of two phases. In the first phase, we generate a reusable plan that determines the opportunity for reuse. The plan is based on a novel reuse discovery mechanism that can identify available results to prevent redundant computation. In the second phase, the plan is executed to produce the subgraph enumeration results. This processing is based on a newly designed reusable parallel search strategy that can efficiently maintain and retrieve the results of set intersection operations. Our implementation on GPUs shows that our approach can achieve up to 5 times speedups compared with the state-of-the-art GPU solutions.

Index Terms—Subgraph enumeration, GPU, reuse

1 Introduction

Given a pattern graph P and a data graph G, subgraph enumeration is to find all subgraphs in G that are isomorphic to P. We refer to the matched subgraph as the *instance*. We also call the subgraph of G that matches a subgraph of P as the *partial instance*. Finding instances of a pattern is important in various domains, such as network motif discovery [1], [2], community detection [3], [4], and frequent subgraph mining [5], [6].

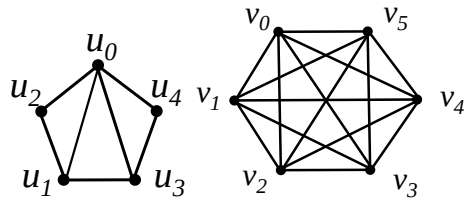


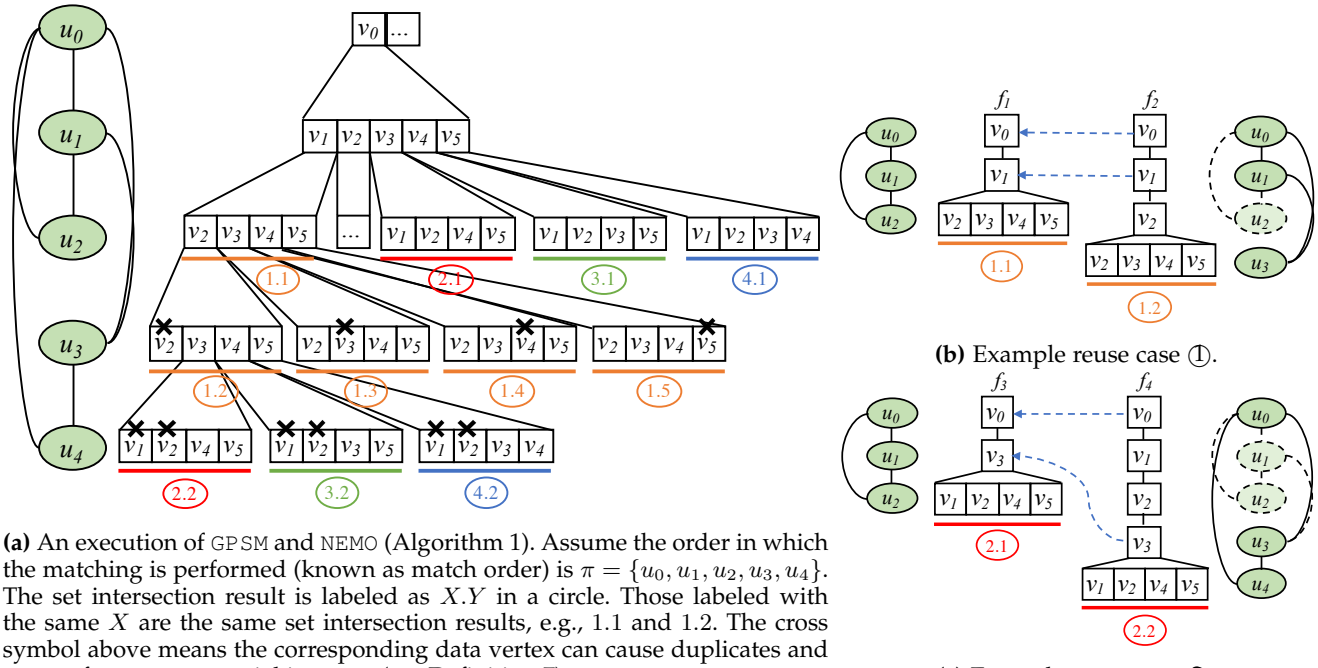
Fig. 1: An example pattern (left) and data graph (right).

1.1 The Case of Reuse

Due to its great importance, subgraph enumeration has been extensively researched. While some works have targeted at the sequential setting [2], [7], [8], [9], [10], others have focused on parallel solutions [11], [12], [13], [14], [15], [16], [17]. More recently, there have been interests [18], [19] in utilizing graphics processing units (GPU) to accelerate subgraph enumeration. These GPU-based solutions [18], [19] iteratively extend partial instances to match one vertex of P at a time until the instances are found. In each iteration, given the partial instances enumerated, we determine a set of candidate vertices of G that match a vertex of P, which is called *the candidate set*. To compute the candidate set, we perform the set intersection operation on the adjacent lists of the relevant vertices of the partial instances. Since the candidate set is computed on the generation of each partial instance, the set intersection operation becomes a bottleneck and takes up most of the running time. Our preliminary study reveals that a large portion of the set intersection operations is redundant. This is because when we compute the candidate set for different vertices of P, the same set intersection operation may be executed on the same adjacent lists. The following example illustrates these redundant set intersection operations.

Example 1. Figure 1 shows the pattern and data graph which are used in our running example. Figure 2 illustrates subgraph enumeration process using the state-of-the-art approaches, i.e., `GPSM` and `NEMO`. Consider the computation for the candidate set of u_2. There is a partial instance f_1 that maps $\{u_0, u_1\}$ to $\{v_0, v_1\}$, denoted as $f_1 = \{(u_0, v_0), (u_1, v_1)\}$. Given f_1, as u_2 is adjacent to u_0 and u_1, we compute the candidate set of u_2 by intersecting the adjacent lists of v_0 and v_1. This computation result is labeled as 1.1 in Figure 2a. Then we consider the computation for the candidate set of u_3. Given the partial instance $f_2 = \{(u_0, v_0), (u_1, v_1), (u_2, v_2)\}$, we compute the candidate set of u_3 by intersecting the adjacent lists of v_0 and v_1 again. This result is labeled as 1.2. As shown in Figure 2b, the results of 1.1 and 1.2 are actually the same. The same redundant results can also be found in 1.3, 1.4, and 1.5 in Figure 2a.

Using the same logic, we can also observe the redundant computation for the candidate set of u_4 (with respect to u_2). Given the partial instance $f_3 = \{(u_0, v_0), (u_1, v_3)\}$, the candidate set of u_2 is computed by intersecting the adjacent lists of v_0 and v_3. The result is labeled as 2.1. Given the partial instance $f_4 = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)\}$, the candidate set of u_4 is computed by intersecting the adjacent lists of v_0



(a) An execution of GPSM and NEMO (Algorithm 1). Assume the order in which the matching is performed (known as match order) is $\pi = \{u_0, u_1, u_2, u_3, u_4\}$. The set intersection result is labeled as $X.Y$ in a circle. Those labeled with the same X are the same set intersection results, e.g., 1.1 and 1.2. The cross symbol above means the corresponding data vertex can cause duplicates and cannot form a new partial instance (see Definition 7).

(b) Example reuse case ①.

(c) Example reuse case ②.

Fig. 2: The case of reuse using example graphs in Figure 1. In Figure 2b and 2c, the dashed lines between partial instances indicate the transformation functions used by our scheme; in each example reuse case, the pattern subgraph on the right is subgraph isomorphic to the pattern subgraph on the left.

and v_3 again. This result is labeled as 2.2. As shown in Figure 2c, the results of 2.1 and 2.2 are actually the same.

In Figure 2a, we also indicate other redundant set intersection operations, e..g, 3.1 and 3.2, 4.1 and 4.2.

The redundant set intersection operation is a significant bottleneck for subgraph enumeration. Our profiling results in Section 6.3 show that the set intersection operations can take up to 95% of the total processing time for subgraph enumeration (Figure 9), while the redundant set intersection operations can be over 99% (Figure 10).

This excessive number of redundant set intersection operations offers an opportunity to further optimize subgraph enumeration. A natural idea is to cache the results of distinct set intersection operations and reuse them to avoid repeated computations. A naïve solution is to implement a centralized key-value store (or any data structure) to maintain a mapping from the relevant vertices $v_1, v_2 \cdots v_n$ (as the key) to the intersection result of their adjacent lists (as the value). In this way, whenever we intersect the adjacent lists of the same vertices $v_1, v_2 \cdots v_n$, we can query the key-value store to retrieve the cached result directly. However, maintaining a key-value store can incur large synchronization overhead when the data structure is concurrently updated by parallel threads [20], [21], [22], [23], [24]. For example, in Figure 2a, a number of threads may concurrently store the results of $1.1, 2.1, 3.1, 4.1$ into the centralized key-value store after computing the candidate set of u_2. This design may lead to even worse performance than the case without exploiting reuse (Figure 8)!

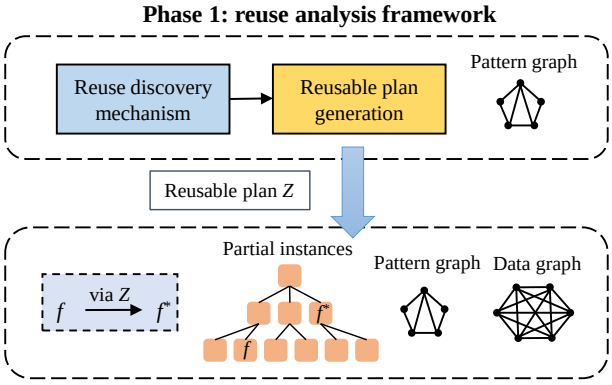


Fig. 3: The two-phase approach to exploit reuse.

1.2 Our Contribution

In this paper, we propose a different solution to support reuse. Our intuition is that each set intersection result is generated for a specific partial instance, and thus the mapping from the partial instance to its corresponding result is obtained "for free" as a by-product. If we know the corresponding partial instance of a set intersection result, we can utilize this mapping to reuse it without maintaining an additional key-value store. However, this idea poses two great challenges. First, given a partial instance f to compute the candidate set, it is unclear how to find another partial instance f^* for which we have computed the set intersection result in the past. Second, even if such a partial instance f^* exists, it is essential to search f^* efficiently at runtime to keep the overheads manageable in order to enjoy the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3035564, IEEE Transactions on Knowledge and Data Engineering

benefits of reuse. To address these challenges, we design a two-phase scheme as shown in Figure 3.

In the first phase, we rely on the **reuse analysis framework** to generate a reusable plan *before execution*. This plan can guide the set intersection operation at runtime to identify the available partial instances that correspond to the results for reuse. The crux of this phase is the reuse discovery mechanism that can establish a transformation function from an existing known partial instance f to another valid partial instance f^* . f^* is guaranteed to correspond to an available result that is helpful to compute the candidate vertices for f. To establish the transformation function, an important condition is that the pattern graph P_n^* matched by f^* and its set intersection result, is subgraph isomorphic to the pattern graph P_n matched by f and its corresponding candidate set. This condition assures the existence of f^*: now that we are given f to comptue the candiate set by following the constraints enforced by P_n, we must have computed the set intersection result for f^* before by following the constraints by P_n^*. Take the reuse case in Figure 2c as an example, given f_4 to compute the result 2.2, the transformation function indicated by the dashed lines will convert f_4 to f_3 such that its corresponding result 2.1 can be reused. In this case, the pattern graph P_n^* matched by f_3 and its corresponding result is a triangle formed by u_0, u_1, u_2; the pattern graph P_n matched by f_4 and the candidate set is a graph formed by u_0, u_1, u_2, u_3, u_4. P_n^* is isomorphic to the triangle counterpart of P_n formed by u_0, u_3, u_4. For each set intersection computaiton, more than one transformation function may be established, as there could be multiple partial instances with available results for reuse. Hence, our reusable plan generation process will choose the best one(s) for the computation of candidate sets.

In the second phase, we design a new parallel approach for subgraph enumeration called **reusable parallel search**, or **RPS** for short, which exploits reuse according to the reusable plan to accelerate the execution. To maintain the result of the set intersection operation, we simply store it along with the partial instance on its generation. To retrieve the result, we follow the reusable plan to generate a partial instance f^*, traverse a light-weight tree structure to find f^*, and then obtain the result stored along with f^* for reuse. To optimize the reuse scope and improve the performance, we conduct the cost analysis to select a reuse-aware match order for RPS. Although our discussion of RPS is mainly based on GPUs, it is a general approach that can be deployed to other parallel architectures such as multi-core CPUs.

We hereby summarize the contributions in this work.

- We devise the reuse analysis framework that facilitates the reuse for parallel subgraph enumeration. It provides a mechanism to systematically discover the cached results available for reuse. It considers all the available reuse options and chooses an efficient execution plan to accelerate the computation.
- We propose the reusable parallel search that can exploit the reuse in execution. RPS is designed to efficiently maintain and retrieve the cached results for reuse. To optimize the reuse scope, it chooses a reuse-aware match order to improve the performance.
- We implement RPS on GPUs and open-source the code at [25]. The experiments show that RPS can achieve up to 5 times speedups over the state-of-the-art solutions for GPUs.

The remaining part of this paper is organized as follows. Section 2 introduces the background of subgraph enumeration, and Section 3 reviews the related works. Section 4 proposes the reuse analysis framework, while Section 5 presents the RPS approach. Section 6 shows the experimental results. Lastly, Section 7 concludes this paper.

2 Preliminary

In this section, we first introduce the preliminary knowledge for subgraph enumeration on GPUs.

2.1 Definitions and Notations

Subgraph enumeration is defined on the *data graph* G and the *pattern graph* P that are both undirected, unlabeled and connected graph. We call the vertices of G and P as the *data vertices* and *pattern vertices* respectively; similarly, we call the edges of G and P as the *data edges* and *pattern edges*.

For a graph g, the vertices and edges of g are denoted as $V(g)$ and $E(g)$. For a vertex v, the adjacent list of v is denoted as $N(v)$. A graph g_1 is a *subgraph* of g_2 if $V(g_1) \subset V(g_2)$ and $E(g_1) \subset E(g_2)$. A graph g_1 is an *induced subgraph* of g_2 on the vertex set $U \subset V(g_2)$ if g_1 has the vertex set $V(g_1) = U$ and the edge set $E(g_1)$ such that $\forall v_1, v_2 \in U$, if $(v_1, v_2) \in E(g_2)$, then $(v_1, v_2) \in E(g_1)$.

Definition 1 (Subgraph Enumeration). A graph g is isomorphic to a pattern graph P, if there exists a bijective mapping $f: V(P) \to V(g)$, such that $(u_1, u_2) \in E(P)$ if and only if $(f(u_1), f(u_2)) \in E(g)$. Given the data graph G and pattern graph P, subgraph enumeration finds all subgraphs of G that are isomorphic to P.

Definition 2 (Match order). Given a pattern graph P, the match order π is a permutation of the pattern vertices $V(P)$ that reflects the order in which $V(P)$ are matched. $\pi(i)$ is the i-th vertex in π. When $1 \leq i \leq j \leq |V(P)|$, $\pi[i:j]$ denotes the set of vertices $\{\pi(k) | i \leq k \leq j\}$. Given π, if $\pi(i) = u$, the position of u on π is $\pi^{-1}(u) = i$.

Definition 3 (Instance / Partial instance). An isomorphism from P to the matched subgraph of G is called an instance of P. Given P and π, the induced subgraph of P on $\pi[1:i]$ is denoted as P_i^π. An instance of P_i^π is a *partial instance* of P. The set of instances of P is denoted as $R(P)$, and the partial instances corresponding to P_i^π is denoted as $R(P_i^\pi)$.

For any partial instance $f \in R(P_i^\pi)$, it is an isomorphic mapping $V(P_i^\pi) \to V(G)$. When a pattern vertex $u \in V(P)$ exists in the domain of f, we say $u \in f$. The projection of f is defined as follows.

Definition 4 (Projection / Prefix / Extension). Given the partial instance $f \in R(P_i^\pi)$, where $1 \leq i \leq |V(P)|$, and a set of pattern vertices $U = \{u | u \in V(P) \wedge u \in f\}$, the *projection* of f on U is the partial instance $f(U) = \{(u, v) | u \in U, v = f(u)\}$. For any i and j with $1 \leq j < i \leq |V(P)|$, given $f_i \in R(P_i^\pi)$, we can have the projection $f_j = f_i(V(P_j^\pi))$. f_j is called the *prefix* of f_i, while f_i is the *extension* of f_j.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3035564, IEEE Transactions on Knowledge and Data Engineering

TABLE 1: Frequently used notations.

Symbol	Descriptions
P, G	pattern graph and data graph
$V(g), E(g)$	vertex set and edge set of g
$N(v)$	adjacent list of v
$R(P)$	the set of instances of P
π	match order
$N_+^\pi(u)$	backward neighbors of u given match order π
P_i^π	induced subgraph of P on the vertex set $\pi[1:i]$
$f, f_\perp$	isomorphic mapping, empty mapping
$f(u)$	the data vertex mapped to u
$f(U)$	the projection of f on the vertex set U
$C(u\|f)$	candidate set for u given partial instance f
$C^*(u\|f)$	result set for u given partial instance f
$u_j \sqsubseteq_g^\pi u_i$	given π, u_i constraint subsumes u_j with constraint subsume function g
T	subsume transform function
$Z = (Z_1, Z_2)$	reusable plan

Definition 5 (Backward neighbor). Given P and π, the backward neighbors of $u \in V(P)$ are the neighbors that are matched before u. Formally, the backward neighbors of u are $N_+^\pi(u) = \{u_n | u_n \in N(u) \wedge \pi^{-1}(u_n) < \pi^{-1}(u)\}$.

Definition 6 (Candidate set). Given P, G, π and the partial instance $f \in R(P_i^\pi)$, the candidate set of $u \in V(P)$ is the set of data vertices $C(u|f) = \{v | v \in V(G) \wedge (\forall u_n \in N_+^\pi(u), (f(u_n), v) \in E(G))\}$. $C(\pi(1)|f) = V(G)$.

Definition 7 (Result set). Given $P, G, \pi, u = \pi(i)$, and the partial instance $f \in R(P_{i-1}^\pi)$, the result set of u is the set of data vertices $C^*(u|f) = \{v | v \in C(u|f) \wedge (\forall u_n \in \pi[1 : i-1], v \neq f(u_n))\}$. $C^*(\pi(1)|f) = V(G)$.

Given P and π, we specifically define P_0^π as a graph with $V(P_0^\pi) = \emptyset$ and $E(P_0^\pi) = \emptyset$. Let the empty isomorphic mapping be $f_\perp = \emptyset$. We define $R(P_0^\pi) = \{f_\perp\}$ to indicate that only $f_\perp$ can be mapped to P_0^π. Since $\pi(1)$ has no backward neighbors, given the empty mapping $f_\perp$, we have $C(\pi(1)|f_\perp) = C^*(\pi(1)|f_\perp) = V(G)$.

For simplicity of illustration, we make the subscript of u_i the same as u_i's position on π, i.e., $\pi(i) = u_i$ where $1 \leq i \leq |V(P)|$. Like previous works [8], [16], [18], [26] we use a connected match order, since it usually achieves better performance than any unconnected match order. This means that P_i^π is a connected graph for $i \in [1, |V(P)|]$ and $N_+^\pi(u_i) \neq \emptyset$ for $i \in (1, |V(P)|]$. Throughput the paper, we consider storing the data graph in the compressed sparse row format (CSR) [27], [28], which is a widely used storage format for efficient graph processing.

Example 2. In Figure 1, a match order for the pattern graph P is $\pi = (u_0, u_1, u_2, u_3, u_4)$. The backward neighbors for each pattern vertex are as follows. $N_+^\pi(u_0) = \emptyset$. $N_+^\pi(u_1) = \{u_0\}$. $N_+^\pi(u_2) = \{u_0, u_1\}$. $N_+^\pi(u_3) = \{u_0, u_1\}$. $N_+^\pi(u_4) = \{u_0, u_3\}$. One of the instances matched to P is $\{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4)\}$. One of the partial instances matched to P_3^π is $f = \{(u_0, v_0), (u_1, v_1), (u_2, v_2)\}$. Given f to compute the candidate set of u_3, we have $C(u_3|f) = \{v_2, v_3, v_4, v_5\}$. To compute the result set of u_3, we have $C^*(u_3|f) = \{v_3, v_4, v_5\}$. v_2 is not in $C^*(u_3|f)$ because f has already mapped u_2 to v_2. With f and $C^*(u_3|f)$, we generate three new partial instances matched to P_4^π: $\{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)\}$, $\{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_4)\}$, $\{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_5)\}$. They are the extensions of f, while f is their prefix. Given the vertex set $V(P_2^\pi)$, the projection of f on $V(P_2^\pi)$ is $f(V(P_2^\pi)) = \{(u_0, v_0), (u_1, v_1)\}$.

Algorithm 1 SubgEnum BFS

Input: the pattern graph P, data graph G and match order π with $\pi(i) = u_i$
Output: the instances $R(P)$

```
1:  R(P_1^π) = {(u_1, v) | v ∈ V(G)}
2:  for 2 ≤ i ≤ |V(P)|
3:      C* = COMPUTE(u_i, R(P_{i-1}^π))
4:      R(P_i^π) = MATERIALIZE (u_i, C*, R(P_{i-1}^π))

5:  procedure COMPUTE(u_i, R(P_{i-1}^π))
6:      parallel for f ∈ R(P_{i-1}^π)
7:          C(u_i|f) = ∩_{∀u∈N_+^π(u_i), v=f(u)} N(v)
8:          C*(u_i|f) = C(u_i|f) − {f(u) | u ∈ f}
9:      Return C*
10: procedure MATERIALIZE(u_i, C*, R(P_{i-1}^π))
11:     parallel for f ∈ R(P_{i-1}^π), F = ∅
12:         parallel for v ∈ C*(u_i|f)
13:             f_n = f ∪ (u_i, v), F = F ∪ f_n
14:         R(P_i^π) = R(P_i^π) ∪ F
15:     Return R(P_i^π)
```

We introduce several propositions as follows. Proposition 1 shows that for the partial instance f matched to P_{i-1}^π and the data vertex v in the result set of u_i, we can add the mapping (u_i, v) into f to generate a new partial instance $f^* = f \cup (u_i, v)$ that is matched to P_i^π. Proposition 2 illustrates that for the partial instance f_i matched to P_i^π and its projection f_{i-1} on the pattern vertices matched before u_i, the data vertex $f_i(u_i)$ is in the result set of u_i given f_{i-1}.

Proposition 1. Given $1 \leq i \leq |V(P)|$, $\forall f \in R(P_{i-1}^\pi), \forall v \in C^*(u_i|f), (f \cup (u_i, v)) \in R(P_i^\pi)$.

Proposition 2. Given $1 \leq i \leq |V(P)|$, $\forall f_i \in R(P_i^\pi), f_{i-1} = f_i(V(P_{i-1}^\pi))$, there is $f_i(u_i) \in C^*(u_i|f_{i-1})$.

We end this subsection with Table 1 listing the frequently used notations throughout the paper.

2.2 BFS Approach

The state-of-the-art GPU solutions (NEMO [18] and GPSM [19]) are based on the breadth-first search (BFS) approach as shown in Algorithm 1. It initializes the partial instances with all data vertices. For each remaining pattern vertex in π, it executes two procedures, namely COMPUTE and MATERIALIZE. COMPUTE iterates each partial instance $f \in R(P_{i-1}^\pi)$ to compute the candidate set $C(u_i|f)$ and the result set $C^*(u_i|f)$. The candidate set computation is the set intersection operation on the relevant adjacent lists (Line 7), which is the runtime bottleneck in execution. To implement such an operation, GPSM adopts the binary search method [19], [29], [30], [31], [32]. Given multiple sets for intersection, it

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3035564, IEEE Transactions on Knowledge and Data Engineering

chooses the set that has the smallest cardinality. Then, it iterates each element in this set to check whether the element exists in all remaining sets via the binary search. After finishing COMPUTE, we obtain the result set C^* for each existing partial instance. Using C^*, MATERIALIZE extends the existing partial instances for P^π_{i-1} to match one more pattern vertex and generate the partial instances for P^π_i.

3 Related Work

Subgraph enumeration algorithms. Based on the approach proposed by Ullman et al. [33], existing algorithms make improvements by various pruning heuristics [2], [7], [8], [9], [10] and selective match orders [7], [8], [26], [34]. While these works reduce the search space mostly by minimizing the number of partial instances searched, our work takes a different approach, i.e., decreasing the computation overhead of generating each partial instance by avoiding unnecessary set intersection operations.

Parallel subgraph enumeration. Parallel solutions for subgraph enumeration have been proposed in different settings, including multi-core CPUs [16], [17], [35], GPUs [18], [19], [36] and distributed clusters [11], [12], [13], [14], [37], [38]. Some multi-core CPU solutions adopt the DFS approach as it is memory-efficient and easy to implement. However, the backtracking nature of DFS approach makes it inefficient on massively parallel architectures. For instance, it can cause irregular execution paths among parallel threads and lead to severe warp divergence on GPUs. As such, most GPU [18], [19] and distributed [11], [12], [13], [14], [37], [38] solutions follow the BFS approach.

Recent graph mining systems including Automine [39] and Pangolin [40] provide programming abstractions to ease users from identifying pattern-specific optimizations e.g., filtering results beforehand and avoiding unnecessary computation. However, these systems do not consider the reuse scenario discussed in this paper.

GPU subgraph enumeration. GPUs have seen recent interests in accelerating graph applications [41], [42], [43], [44]. To optimize GPU-based graph applications, [45] explores various designs for some morph algorithms where the graph structures can change in an unpredictable fashion. However, in subgraph enumeration, the graphs, i.e., the intermediate result, change in a regular way as per the pattern graph.

For GPU subgraph enumeration [18], [19], [36], they follow the same BFS approach as in Algorithm 1 and only slightly differ in implementation details. For set intersection operations, `NEMO` checks whether a data vertex exists in another adjacent list by a linear scan, while `NEMO` and `GSI` adopt the binary search. To materialize set intersection results, `NEMO` and `GPSM` perform the computation in one additional time to obtain the result size, but `GSI` pre-allocates the memory for writes and compact the result afterwards. Besides some implementation strategies, `GPSM` and `GSI` also target general subgraph matching, i.e., subgraph enumeration on labeled graphs, and propose specific optimizations, e.g., PCSR structures in `GSI` to efficiently filter the data vertices matched to a label. These optimizations do not apply in our context as subgraph enumeration is defined on unlabeled graphs. Desipte the differences in implementation, these works can apply our reuse scheme to boost the performance as they are based on the same algorithm.

Set intersection operations. There are many works [46], [47], [48], [49] proposed in the CPU context to improve the single-thread performance for set intersection operations. These works may not be efficiently deployed on GPUs, because they do not consider load balance among parallel threads, warp divergence and coalesced memory accesses to optimize the performance. Green et al. [50] proposes a merge-based approach for set intersection operations on GPUs. It partitions two sorted arrays into non-overlapping and equal chunks, each of which can be independently proccessed by a group of threads. However, recent work [29] shows that the simple binary search [19], [30] is more efficent because it actually reduces the warp divergence and increases the coalesced memory accesses.

Reuse in subgraph enumeration. To our knowledge, there are two works [8], [16] that propose techniques similar to reuse for subgraph enumeration, but their reuse scope is limited. To compute the candidate set of a pattern vertex u_i, LIGHT [16] can utilize the candidate set of a pattern vertex u_j with $j < i$ that is previously computed, only when $N^\pi_+(u_j) \subseteq N^\pi_+(u_i)$. In comparison, our approach can reuse the computed result even when $N^\pi_+(u_j) \not\subset N^\pi_+(u_i)$ (see Example 5). TurboIso [8] proposes the COMB/PERM strategy to merge "equivalent" pattern vertices (i.e., pattern vertices have exactly the same neighbors) into one condensed pattern vertex, so that the candidate data vertices can be generated only once but shared among the equivalent pattern vertices. However, the equivalence of pattern vertices greatly restricts the application scope of the scheme. Besides the limited reuse scope, LIGHT and TurboIso are based on the DFS approach, which is difficult to be efficiently parallelized on GPUs as we mention earlier.

4 Reuse Analysis Framework

This section introduces the reuse analysis framework that discovers the reuse opportunities and generates an efficient execution plan for the candidate set computation.

4.1 Overview

To reuse the cached results and avoid the redundant computation, given the partial instance f to compute the candidate set $C(u_i|f)$, we seek to find a partial instance f^* that was generated before and a pattern vertex u_j such that the corresponding candidate set $C(u_j|f^*)$ is useful for computing $C(u_i|f)$.

There are two challenges to reuse the cached result $C(u_j|f^*)$ as mentioned above. First, since there are multiple available pattern vertices and a tremendous number of partial instances generated in the past, it is unclear how to identify a specific u_j and f^* such that the corresponding cached result $C(u_j|f^*)$ is useful. Second, even when we can identify the cached results, we might not be able to find a cached result $C(u_j|f^*)$ that is exactly equal to $C(u_i|f)$. In this case, it is uncertain how to make use of the available cached results to accelerate the computation of $C(u_i|f)$.

To address these challenges, our reuse analysis framework provides two components, namely reuse discovery mechanism and reusable plan generation. The reuse discovery mechanism inspects the pattern vertex u_j and establishes a transformation function to generate the partial

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3035564, IEEE Transactions on Knowledge and Data Engineering

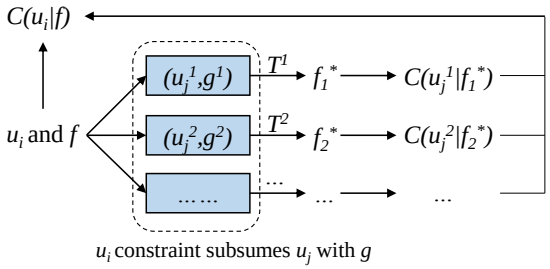


Fig. 4: Overview of reuse discovery mechanism.

instance f^* such that the cached result $C(u_j|f^*)$ is available at runtime for reuse. With the available cached results, the reusable plan generation evaluates each reuse option, and then produces a reuse-aware execution plan for the candidate set computation.

4.2 Reuse Discovery Mechanism

Our mechanism is based on an important relationship among two pattern vertices called *constraint subsume*. It is an indication that the candidate set computation of u_i can reuse the candidate set of u_j. When u_i constraint subsumes u_j, P_j^π is isomorphic to a subgraph of P_i^π with a special mapping g, which specifically maps u_j to u_i and preserves the orders among the pattern vertices in P_j^π. The constraint subsume relationship is formally defined as follows.

Definition 8 (Constraint subsume). Let $1 \leq j < i \leq |V(P)|$. Given π, u_i constraint subsumes u_j with g, denoted as $u_j \sqsubseteq_g^\pi u_i$, if there exists a mapping $g\colon V(P_j^\pi) \to V(P_i^\pi)$ such that (1) $g(u_j) = u_i$; (2) $\forall u \in V(P_j^\pi), \forall u_n \in N_+^\pi(u), g(u_n) \in N_+^\pi(g(u))$; (3) $\forall k, w \in [1, j]$, if $k < w$, then $\pi^{-1}(g(u_k)) < \pi^{-1}(g(u_w))$. g is called the constraint subsume function.

With the constraint subsume function g, which is a mapping between the pattern vertices, we define the subsume transform function T that employs g to convert the partial instance f to a new partial instance f^*. f^* would map any vertex in its domain $u \in f^*$ to the same data vertex as f maps $g(u)$ to, i.e., $f^*(u) = f(g(u))$.

Definition 9 (Subsume transform function). Given a partial instance f, a constraint subsume function g and a set of pattern vertices $U \subseteq V(P)$, for any $u \in U$, u is in the domain of g and $g(u)$ is in the domain of f. The subsume transform function T generates a new partial instance as

$$T(f, g, U) = \{(u, f(g(u)))|u \in U\}$$

Example 3. For the example pattern graph in Figure 1, assume the match order $\pi = (u_0, u_1, u_2, u_3, u_4)$. u_3 constraint subsumes u_2 with g_1. g_1 maps (u_0, u_1, u_2) to (u_0, u_1, u_3). u_4 constraint subsumes u_2 with g_2. g_2 maps (u_0, u_1, u_2) to (u_0, u_3, u_4). Given the partial instance $f_1 = \{(u_0, v_0), (u_1, v_1), (u_2, v_2)\}$, we use g_1 and $V(P_2^\pi)$ to apply the subsume transform function $T(f_1, g_1, V(P_2^\pi)) = \{(u_0, v_0), (u_1, v_1)\}$. Given the partial instance $f_2 = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)\}$, we use g_2 and $V(P_2^\pi)$ to apply the subsume transform function $T(f_2, g_2, V(P_2^\pi)) = \{(u_0, v_0), (u_1, v_3)\}$.

Mechanism overview. Figure 4 illustrates how the reuse discovery mechanism identifies the available cached results. For the candidate set computation of u_i, our mechanism finds all possible pairs of (u_j, g) such that u_i constraint subsumes u_j with the constraint subsume function g. All the pairs can be generated by enumerating each pattern vertex u_j with $j < i$ and any possible mappings g between the pattern vertices. As the constraint subsume relationship only relies on the pattern graph P and match order π, all the pairs can be generated before enumerating the subgraphs of the data graph. By means of the pair (u_j, g), given any partial instance $f \in R(P_{i-1}^\pi)$ that has matched the pattern vertices before u_i, we can apply the subsume transform function T on the vertex set $V(P_{j-1}^\pi)$ to generate a new partial instance $f^* = T(f, g, V(P_{j-1}^\pi))$. Then the corresponding candidate set $C(u_j|f^*)$ can be reused for $C(u_i|f)$.

Benefits of reuse. To understand the benefit of reusing $C(u_j|f^*)$, assuming the generated partial instance f^* exists, Lemma 1 proves that $C(u_j|f^*)$ is the partial execution result to compute $C(u_i|f)$. Specifically, it is the intersection result over the adjacent lists of the data vertices matched to *some* backward neighbors of u_i. As $C(u_i|f)$ is the intersection result corresponding to all backward neighbors of u_i, $C(u_j|f^*)$ is a portion of workload necessary to compute $C(u_i|f)$, and reusing $C(u_j|f^*)$ for $C(u_i|f)$ can save the efforts of this portion. By Lemma 1, there is always $C(u_i|f) \subseteq C(u_j|f^*)$. When $C(u_j|f^*)$ is the intersection result that covers all backward neighbors of u_i, then $C(u_i|f) = C(u_j|f^*)$ and hence we can directly use $C(u_j|f^*)$ as the result of $C(u_i|f)$. For the case $C(u_i|f) \neq C(u_j|f^*)$, because $C(u_i|f) \subset C(u_j|f^*)$, we can still reuse $C(u_j|f^*)$ but need further computation to get $C(u_i|f)$ (more details are discussed in the next subsection).

Lemma 1. Given π, assume $u_j \sqsubseteq_g^\pi u_i$. Given the partial instance $f \in R(P_{i-1}^\pi)$, $f^* = T(f, g, V(P_{j-1}^\pi))$. If $f^* \in R(P_{j-1}^\pi)$, then (1) there exists $U \subseteq N_+^\pi(u_i)$ such that $C(u_j|f^*) = V(G) \cap \{\cap_{\forall u \in U} N(f(u))\}$; (2) $C(u_i|f) \subseteq C(u_j|f^*)$.

Proof: This lemma can be proved using Lemma 2.

Feasibility of reuse. Although we can identify the cached result $C(u_j|f^*)$ that is beneficial for reuse, reusing $C(u_j|f^*)$ depends on an important condition that the generated partial instance f^* exists, which is assumed to be true by Lemma 1. In fact, the existence of f^* is guaranteed by the constraint subsume relationship. The intuition is that when u_i constraint subsumes u_j with the constraint subsume function g, there is an isomorphism $g' = g - (u_j, u_i)$ from P_{j-1}^π to a subgraph of P_{i-1}^π. We denote this isomorphic subgraph of P_{i-1}^π as p_{i-1}. Given the partial instance $f \in R(P_{i-1}^\pi)$, the projection $f(V(p_{i-1}))$ is a valid match to the pattern p_{i-1}. Since the pattern P_{j-1}^π is structurally the same as p_{i-1} except for the different vertices, the projection $f(V(p_{i-1}))$ can also be a match to the pattern P_{j-1}^π if we can arrange the domain of $f(V(p_{i-1}))$ with the pattern vertices $V(P_{j-1}^\pi)$. Such an arrangement is actually the application of the subsume transform function T. As such, the partial instance f^* generated by T is a match to the pattern P_{j-1}^π and thus f^* exists in $R(P_{j-1}^\pi)$.

To prove the existence of f^*, we first introduce Lemma 2. Assuming f^* exists, this lemma proves that for any $u_k \in$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3035564, IEEE Transactions on Knowledge and Data Engineering

$V(P_j^\pi)$ and the pattern vertex $g(u_k)$ mapped by g, given any partial instance f that has matched the pattern vertices before $g(u_k)$ and the partial instance f^* generated by T, there are subsume relationships for the candidate set and result set, i.e., $C(g(u_k)|f) \subseteq C(u_k|f^*)$ and $C^*(g(u_k)|f) \subseteq C^*(u_k|f^*)$. Assisted by Lemma 2, Theorem 1 proves that given any partial instance $f \in R(P_{i-1}^\pi)$, the constraint subsume relationship can guarantee the existence of f^*, which is generated by the subsume transform function T.

Lemma 2. Given π, assume $u_j \sqsubseteq_g^\pi u_i$. Given $u_k \in V(P_j^\pi)$, we define $q(k) = \pi^{-1}(g(u_k)) - 1$. $\forall u_k \in V(P_j^\pi), \forall f \in R(P_{q(k)}^\pi)$, $f^* = T(f, g, V(P_{k-1}^\pi))$, if $f^* \in R(P_{k-1}^\pi)$, then $C(g(u_k)|f) \subseteq C(u_k|f^*)$ and $C^*(g(u_k)|f) \subseteq C^*(u_k|f^*)$.

Proof: Since $u_j \sqsubseteq_g^\pi u_i$, $\forall u \in N_+^\pi(u_k), g(u) \in N_+^\pi(g(u_k))$, which is an important condition for $C(g(u_k)|f) \subseteq C(u_k|f^*)$. Due to space limitations, the proof is omitted.

Theorem 1. Given π, assume $u_j \sqsubseteq_g^\pi u_i$. $\forall f \in R(P_{i-1}^\pi)$, $f^* = T(f, g, V(P_{j-1}^\pi))$, there exists $f^* \in R(P_{j-1}^\pi)$.

Proof: We can prove by induction to show that given u_k, for any $f \in R(P_{b(k)}^\pi)$ and $f^* = T(f, g, V(P_k^\pi))$, we have $f^* \in R(P_k^\pi)$. Please refer to supplementary materials for details.

Example 4. As shown by Figure 2a, there is a partial instance $f = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)\} \in R(P_4^\pi)$. Continuing Example 3, we have u_4 constraint subsumes u_2 with g_2. Let $f^* = T(f, g_2, V(P_2^\pi)) = \{(u_0, v_0), (u_1, v_3)\}$. As shown in Figure 2a, $f^* \in R(P_2^\pi)$, which is a use case of Theorem 1.

4.3 Reusable Plan Generation

Although the available cached results can be identified by the reuse discovery mechanism, it is not easy to reuse them as we may not be able to find a cached result that is exactly equal to the candidate set we are computing. In that case, it is unclear how to choose a set of cached results for the candidate set computation so as to produce the correct execution result and optimize the runtime performance.

To solve this problem, we design an execution plan called *reusable plan*. The plan is composed by a series of sub-plans, and each sub-plan corresponds to the candidate set computation of one pattern vertex. We call the sub-plan as *reusable vertex plan*. As shown by Figure 5, the reusable vertex plan $Z(u_i) = (Z_1(u_i), Z_2(u_i))$ consists of two parts: $Z_2(u_i)$ is a subset of the backward neighbors of u_i, while $Z_1(u_i)$ refers to a set of cached results that can be located by the reuse discovery mechanism. Specifically, $Z_1(u_i)$ comprises a series of (u_j, g), where u_i constraint subsumes u_j with the constraint subsume function g. According to the reuse discovery mechanism, given any partial instance $f \in R(P_{i-1}^\pi)$, (u_j, g) can be used to generate a partial instance f^* so that the cached result $C(u_j|f^*)$ can be reused. By Lemma 1, $C(u_j|f^*)$ is the intersection result over the adjacent lists of the data vertices matched to some backward neighbors of u_i. We say this set of backward neighbors is covered by (u_j, g). An important requirement of the reusable vertex plan is to have the pattern vertices covered by each (u_j, g) in $Z_1(u_i)$ and the pattern vertices $Z_2(u_i)$ constitute $N_+^\pi(u_i)$.

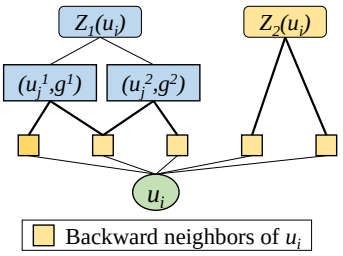


Fig. 5: Illustration of reusable vertex plan $Z(u_i)$.

Definition 10 (Reusable vertex plan). The reusable vertex plan for u_i is defined as $Z(u_i) = (Z_1(u_i), Z_2(u_i))$ that satisfies: (1) for each $(u_j, g) \in Z_1(u_i)$, there is $u_j \sqsubseteq_g^\pi u_i$; (2) $Z_2(u_i) \subseteq N_+^\pi(u_i)$; (3) $\{u|(u_j, g) \in Z_1(u_i), u_n \in N_+^\pi(u_j), u = g(u_n)\} \cup Z_2(u_i) = N_+^\pi(u_i)$.

Execution by reusable plan. Following the reusable vertex plan $Z(u_i)$, the candidate set computation would intersect the cached results that are discovered with each $(u_j, g) \in Z_1(u_i)$ and the adjacent lists of the data vertices matched to the pattern vertices $Z_2(u_i)$ (i.e., Equation 1). Theorem 2 proves such a method leads to the correct result.

Theorem 2. Given the reusable vertex plan $Z(u_i) = (Z_1(u_i), Z_2(u_i))$ and the partial instance $f \in R(P_{i-1}^\pi)$, the candidate set of u_i can be computed by

$$L(u_i|f) = (\cap_{\forall (u_j,g) \in Z_1(u_i), f^* = T(f,g,V(P_{j-1}^\pi))} C(u_j|f^*)) \cap (\cap_{\forall u \in Z_2(u_i)} N(f(u))) \cap V(G) \quad (1)$$

Proof: Please refer to the supplemental materials.

Example 5. From Example 3, u_4 constraint subsumes u_2 with g_2. A reusable vertex plan for u_4 can be $Z(u_4) = (Z_1(u_4), Z_2(u_4))$, where $Z_1(u_4) = \{(u_2, g_2)\}$ and $Z_2(u_4) = \emptyset$. From Example 4, we have a partial instance $f = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)\} \in R(P_4^\pi)$ and $f^* = \{(u_0, v_0), (u_1, v_3)\} \in R(P_2^\pi)$. Following Equation 1 to compute $C(u_4|f)$, $L(u_4|f) = C(u_2|f^*)$. As shown in Figure 2a, $C(u_2|f^*) = \{v_1, v_2, v_4, v_5\}$, which we reuse for $C(u_4|f)$. Thus, $C(u_4|f) = L(u_4|f) = \{v_1, v_2, v_4, v_5\}$.

Plan generation. Since the plan for each vertex is independent, we can generate it separately. According to Equation 1, the number of intersection operations performed is $|Z_1(u_i)| + |Z_2(u_i)| - 1$, and thus an efficient plan should reduce the cardinality of $Z_1(u_i)$ and $Z_2(u_i)$. As shown in Figure 5, the requirement of $Z(u_i)$ is that the backward neighbors $N_+^\pi(u_i)$ shall be fully covered by each $(u_j, g) \in Z_1(u_i)$ and the pattern vertices $Z_2(u_i)$. Therefore, we formalize the plan generation as a minimum set cover problem. We construct a set S containing all collections of pattern vertices covered by the possible elements in $Z_1(u_i)$ and $Z_2(u_i)$. Then we find a minimum subset $S' \subseteq S$ that can cover the universe $N_+^\pi(u_i)$. Due to space limitation, we leave the details of plan generation process in supplementary materials.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3035564, IEEE Transactions on Knowledge and Data Engineering

5 Reusable Parallel Search

This section presents our reusable parallel search (RPS) at runtime to make use of the available cached results that are identified by the reuse analysis framework before execution.

5.1 Design

RPS is based on BFS, but it takes additional efforts to maintain and retrieve the cached results in execution. The maintenance is easy because we simply materialize the result of a candidate set on its generation. To retrieve the cached result, we can apply the transformation function established by the reuse discovery mechanism to generate the partial instance f^* and use the pattern vertex u_j specified in the reusable vertex plan. Then, in order to acquire the cached result $C(u_j|f^*)$, we need to search f^* in execution.

Index partial instances. We organize the partial instances as a tree structure as they are generated. This facilitates the efficient retrieval of the cached results. Note that for each partial instance $f \in R(P^\pi_{i-1})$, MATERIALIZE generates a set of extension $F \subset R(P^\pi_i)$ for f. We can consider each partial instance as a tree node and place F as the descendants of f in the tree, since f is the common prefix of any partial instance in F. To build up the tree, we create a tree node for each partial instance $f \in R(P^\pi_i)$ where $1 \leq i < |V(P)|$ and arrange its descendants. A root representing $f_\perp$ is also added to serve as the ancestors of all the tree nodes. Thus, the resulting tree has the partial instances for $R(P^\pi_i)$ in the i-th level and $f_\perp$ in the 0-th level.

Search partial instances. With the tree structure, we follow a two-step method to search partial instances. To compute the candidate set of u_i, given a partial instance $f \in R(P^\pi_{i-1})$, we search the corresponding partial instance $f^* \in R(P^\pi_{j-1})$ ($j < i$). Considering each partial instance as a tree node, the search of f^* is the movement on the tree structure from the node f to the node f^*. As such, in the first step of the search, we move from the node f to the lowest common ancestor (LCA) between f and f^*, say f_L; in the second step, we move from f_L to the node f^*. The first step can be easily implemented by iteratively tracking the ancestor of each visited tree node until reaching f_L. In the second step, we need to iteratively search a specific descendant of a tree node to move down the tree until reaching f^*. Note that there could be a number of descendants for a tree node, making the simple linear scan over all the descendants rather slow.

To accelerate the search, we exploit the property that the descendants of a tree node are sorted, which allows us to apply the binary search. This property is true because we adopt CSR to store the data graph. To implement COMPUTE, we can easily materialize the candidate set and result set in sorted order. With the sorted result set, MATERIALIZE generates the sorted extension for each partial instance, which is based on the following defined order.

Definition 11 (Order among partial instances). Assume there is a total order for $V(G)$. Given $f_1, f_2 \in R(P^\pi_i)$ ($1 \leq i \leq |V(P)|$), $f_1 < f_2$ if there exists $k \in [1, i]$ such that $(\forall w \in [1, k), f_1(u_w) = f_2(u_w))$ and $(f_1(u_k) < f_2(u_k))$.

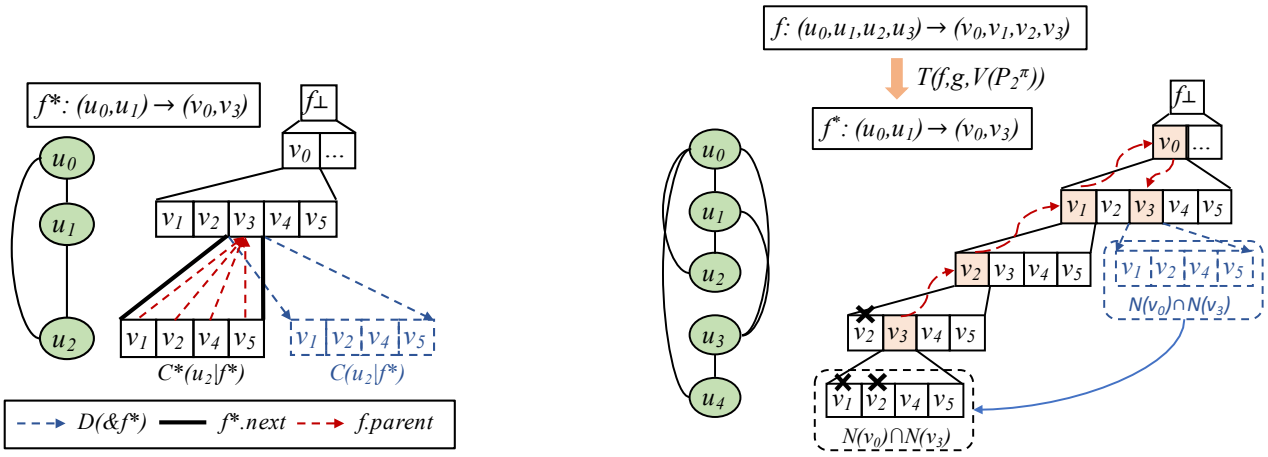
Example 6. Figure 6b shows the tree structure built for the example graph in Figure 1. Assume the match order is $\pi = (u_0, u_1, u_2, u_3, u_4)$. The partial instance $f_0 = \{(u_0, v_0), (u_1, v_1)\}$ has 4 extensions matched to P^π_3, namely $f_1 = \{(u_0, v_0), (u_1, v_1), (u_2, v_2)\}$, $f_2 = \{(u_0, v_0), (u_1, v_1), (u_2, v_3)\}$, $f_3 = \{(u_0, v_0), (u_1, v_1), (u_2, v_4)\}$, $f_4 = \{(u_0, v_0), (u_1, v_1), (u_2, v_5)\}$. These extensions are placed under f_0 as the descendants of f_0. These extensions are sorted based on the order we define, i.e., $f_1 < f_2 < f_3 < f_4$. For the partial instance $f = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)\}$ and $f^* = \{(u_0, v_0), (u_1, v_3)\}$, the LCA between f and f^* is $\{(u_0, v_0)\}$.

Algorithm 2 SubgEnum RPS

Input: the pattern graph P, data graph G, reusable vertex plan of all pattern vertices $Z = (Z_1, Z_2)$, match order π with $\pi(i) = u_i$
Output: the instances $R(P)$
Maintain: $\forall f \in R(P^\pi_i), f.parent, f.next, D(\&f)$

1: **procedure** COMPUTE($u_i, R(P^\pi_{i-1})$)
2: **parallel for** $f \in R(P^\pi_{i-1})$
3: $B = \{N(f(u))|u \in Z_2(u_i)\}$
4: **for** $(u_j, g) \in Z_1(u_i)$ **do**
5: $f^* = T(f, g, V(P^\pi_{j-1}))$
6: $C(u_j|f^*) =$ SEARCH(u_i, u_j, g, f, f^*)
7: $B = B \cup C(u_j|f^*)$
8: $C(u_i|f) = \cap_{\forall B_k \in B} B_k$
9: $D(\&f) = C(u_i|f)$
10: $C^*(u_i|f) = C(u_i|f) - \{f(u)|u \in f\}$
11: **Return** C^*
12: **procedure** MATERIALIZE($u_i, C^*, R(P^\pi_{i-1})$)
13: **parallel for** $f \in R(P^\pi_{i-1}), F = \emptyset$
14: **parallel for** $v \in C^*(u_i|f)$
15: $f_n = f \cup (u_i, v), F = F \cup f_n$
16: $f_n.parent = \&f$
17: $R(P^\pi_i) = R(P^\pi_i) \cup F$
18: $f.next = \&F$
19: **Return** $R(P^\pi_i)$

20: **procedure** SEARCH(u_i, u_j, g, f, f^*)
21: $w \leftarrow argmax_{z \in [1, j-1]} \{u_x = g(u_x), \forall x \in [1, z]\}$
22: **if** w exists **then**
23: $l = i - 1, ptr = \&f$
24: **while** $l > w$ **do**
25: $ptr = (*ptr).parent, l = l - 1$
26: **else**
27: $l = 0, ptr = \&f_\perp$
28: **while** $l < j - 1$ **do**
29: $F = *((*ptr).next)$ ▷ $F \subset R(P^\pi_l)$
30: $f_{l+1} = f^*(V(P^\pi_{l+1}))$
31: $ptr \leftarrow$ Binary search f_{l+1} over F ▷ F is sorted
32: $l = l + 1$
33: $C(u_j|f^*) = D(ptr)$ ▷ Now $*ptr = f^*$
34: **Return** $C(u_j|f^*)$



(a) An execution of MATERIALIZE. Besides materializing partial instances, given f^* as shown above, we maintain the auxiliary tree structure: $f^*.next$ tracks the result set $C^*(u_2|f^*)$; $D(\&f^*)$ records the candidate set $C(u_2|f^*)$, i.e., the set intersection result; for each child f of f^*, $f.parent$ tracks its parent f^*.

(b) An execution of COMPUTE. Suppose the reusable vertex plan has $Z_1(u_4) = \{(u_2, g)\}$ and $Z_2(u_4) = \emptyset$, where g maps (u_0, u_1, u_2) to (u_0, u_3, u_4). Given f as shown above, we would first generate f^*, traverse the tree structure to locate f^*, and then reuse the result of $C(u_2|f^*)$, i.e., $N(v_0) \cap N(v_3)$, for $C(u_4|f)$.

Fig. 6: An example using Algorithm 2 on the example graphs in Figure 1. The match order is $\{u_0, u_1, u_2, u_3, u_4\}$. The cross symbol $\times$ means the corresponding data vertex is in the candidate set but not result set.

5.2 Implementation

Algorithm 2 illustrates the implementation of RPS. We only show the implementation of COMPUTE and MATERIALIZE in Algorithm 2, as the workflow is the same as Algorithm 1. In Algorithm 2, we use reference (&) and dereference ($*$) operator. For a variable a, $\&a$ denotes the memory address of a; for a reference p, $*p$ denotes the object at the address of p. For each partial instance $f \in R(P_i^\pi)$, we maintain the location of the prefix of f by $f.parent$ and the location of the extension of f by $f.next$. $f.parent$ and $f.next$ represent the tree structure used to search the partial instance. We also maintain $D(\&f)$ to track the cached intersection result along with the location of f.

COMPUTE and MATERIALIZE. The COMPUTE procedure differs from that of Algorithm 1 in the candidate set computation. Guided by the reusable vertex plan $Z(u_i)$, it collects the adjacent lists of the data vertices matched to $Z_2(u_i)$ (Line 3) and the cached intersection result indicated in $Z_1(u_i)$ (Lines 4-7). To retrieve the cached intersection result, we first apply the subsume transform function T on the given partial instance f to obtain another partial instance f^* that guarantees $f^* \in R(P_{j-1}^\pi)$ (Line 5). Then, SEARCH is used to find the candidate set $C(u_j|f^*)$ that was computed in the past (Line 6). The collected adjacent lists and cached intersection result are intersected to compute the candidate set (Line 8). After that, the computation result is stored along with f (Line 9). With the result set generated by COMPUTE, similar to Algorithm 1, MATERIALIZE generates a set of extension F for each partial instance f. Additionally, it tracks the extensions by $f.next$ and the prefix of each extension $f_n \in F$ by $f_n.parent$.

SEARCH. Given $f \in R(P_{i-1}^\pi), f^* \in R(P_{j-1}^\pi)$, SEARCH traverses the tree structure from f to get the location of f^*. In the beginning, we identify a consecutive range $[1, z]$ of the domain of g such that g can map each element in $[1, z]$ to itself (Line 21). f and f^* would have the common prefix on such a range. That is, $\forall x \in [1, z], f(u_x) = f^*(u_x)$. Let w be the largest z if it exists. If w does not exist, there is no common prefix between f and f^*. In this case, the LCA between f and f^* is $f_\perp$, i.e., the root of the tree in the 0-th level, so we directly search f^* from $f_\perp$ (Line 27). If w exists, the LCA between f and f^* is in the w-th level. The LCA is the partial instance $f_L = f(\pi[1, w])$. In this case, we initialize the searching position ptr as the location of f in $(i-1)$-th level (Line 23). Then, we follow the pointer to the ancestor of the tree node, $(*ptr).parent$, to reach the LCA in w-th level (Lines 24-25). After ptr can position the LCA, we iteratively move downward the tree to reach f^* (Lines 28-32). In each iteration, we fetch the descendants of a tree node via $(*ptr).next$ (Line 29). Among the descendants F, we are interested in the one that is the projection of f^* on $V(P_{l+1}^\pi)$ (Line 30). To search that descendant, the binary search over F is conducted (Line 31). After reaching the $(j-1)$-th level of the tree, ptr can position f^* and we can fetch the result $C(u_j|f^*)$ by $D(ptr)$.

Memory management. Since GPUs are equipped with limited device memory but there are a large number of partial instances, the execution of RPS is implemented in a pipeline manner similar to [18]. Specifically, we reserve a portion of device memory θ_i for each iteration i. To generate the partial instances for $R(P_i^\pi)$, we extend a subset of partial instances achieved in the previous iteration and produce a subset $F_i \subset R(P_i^\pi)$. The size of F_i and the corresponding cached results are ensured to fit to the reserved memory θ_i. Then F_i would be used in the next iteration to generate the partial instances for $R(P_{i+1}^\pi)$. Due to the pipeline processing, only a subset $F_i \subset R(P_i^\pi)$ and the corresponding cached results are stored in iteration i. When a partial instance to be searched f^* is not in the subset F_i, we are unable to locate the cached result corresponding to f^* and thus the reuse

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3035564, IEEE Transactions on Knowledge and Data Engineering

cannot be exploited. In that case, we simply fall back to the traditional method of candidate set computation without the reuse. In execution, we find that such a case seldom happens and the cached results are available most of the time (with the reuse ratio up to 99% as shown in Figure 10).

Example 7. Figure 6 shows examples of COMPUTE and MATERIALIZE of Algorithm 2. Figure 6a shows the tree structure maintained for the partial instance $f^* = \{(u_0, v_0), (u_1, v_3)\}$. $f^*.next$ tracks the corresponding result set and $f.parent$ records the parent of each tree node. They help to navigate the tree structure in SEARCH. $D(\&f^*)$ records the location of the set intersection result, i.e., $C(u_2|f^*)$.

Figure 6b shows an example of the candidate set computation in COMPUTE. $f = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)\} \in R(P_4^\pi)$ is given to compute the candidate set $C(u_4|f)$. First, we apply the subsume transform function to get $f^* = \{(u_0, v_0), (u_1, v_3)\}$. To search f^*, we traverse the tree as indicated by the arrows. Starting from f, we first go along the ancestors: $\{(u_0, v_0), (u_1, v_1), (u_2, v_2)\}$, $\{(u_0, v_0), (u_1, v_1)\}$, $\{(u_0, v_0)\}$. $\{(u_0, v_0)\}$ is the LCA between f and f^*. Then we go from $\{(u_0, v_0)\}$ to $f^* = \{(u_0, v_0), (u_1, v_3)\}$ by a binary search over the descendants of $\{(u_0, v_0)\}$. After reaching f^*, we retrieve the result aligned with f^*, i.e., $C(u_2|f^*) = \{v_1, v_2, v_4, v_5\}$, which was computed by $N(v_0) \cap N(v_3)$. To reuse this result, we copy it for $C(u_4|f)$. Now that $C(u_4|f)$ is ready, we can proceed to compute the result set $C^*(u_4|f)$.

5.3 Match Order Selection

Although the reuse can be exploited for any given match order π, the order can affect the performance of RPS because the reusable plan can vary with π and thus the reuse scope differs. To evaluate a match order, we rely on the reuse-aware cost analysis that considers the performance gains and overheads of reuse. The analysis differs from the cost models proposed in previous works [8], [10], [18], [19] that do not exploit reuse. Due to space limitation, we leave the discussion of cost analysis in the supplementary materials.

The match order selection works as follows. First, we enumerate each possible order of $V(P)$, say π, and generate the reusable vertex plan as discussed in Section 4.3 for each pattern vertex. Then, our reuse-aware cost analysis evaluates the execution cost given π. In the end, the order with the least cost is chosen. Note that the reuse-aware order selection may choose the plan that does not exploit reuse. In this case, the execution would be the same as the traditional BFS approach (Algorithm 1).

6 EXPERIMENTS

In this section, we first introduce the experimental setups and then report the evaluation results.

6.1 Experimental Setup

Implementation. We implement the following methods on GPUs for evaluation.

- `GPSM` [19]: The state-of-the-art GPU solution based on Algorithm 1. Because the method is originally proposed for labeled graphs, it has a filter phase to generate candidate data vertices by the labels and a join phase to enumerate the matched subgraphs. We only implement the join phase since subgraph enumeration is defined on unlabeled graphs and the filter phase is not needed.
- `NEMO` [18]: The recent GPU solution based on Algorithm 1. As the work [18] is originally designed for network motif discovery, we only implement its component of subgraph enumeration for the experiment. `NEMO` is the same as `GPSM` except that it implements the set intersection operations in a different way (see Section 3 for details).
- `REUSE_KV`: The direct method that uses the key-value store to exploit the reuse. Based on `GPSM`, we implement the hash table by the CUDPP library [51]. When a new result of set intersection operation is generated, it is inserted into the hash table with the key as the relevant data vertices. To perform the set intersection, we would query the hash table to check to reuse the cached results As there are a large number of set intersection results but the GPU memory is rather limited, the hash table will be released if its memory consumption exceeds the pre-defined threshold. We tune the parameter and find setting it to 1GB can achieve good performance. After the memory release, the hash table can continue to store the cached results as usual.
- `RPS`: The implementation of our approach (Algorithm 2) on top of `GPSM`.

Datasets. Statistics of the datasets are shown in Table 2. The datasets include *youtube* (YT), *wiki-Talk* (WK), *wiki-topcats* (TP), *twitter-higgs* (TW), *dogster* (DG) and *orkut* (OR). The datasets are downloaded from SNAP [52] and Network Repository [53].

TABLE 2: The datasets used in the experiments.

Dataset	YT	WK	TP	TW	DG	OR
$\|V\|(\times 10^6)$	1.1	2.3	1.8	0.4	0.4	3.0
$\|E\|(\times 10^6)$	2.9	5.0	28.5	14.9	8.5	117.1

Queries. Figure 7 lists the queries used in the experiments. These queries have been evaluated in a number of recent works [11], [14], [54], [55]. To evaluate the performance, we count the number of instances matched to the query and report the running time.

Experimental Environment. The experiments are conducted on a machine equipped with 128GB main memory, two NVIDIA TITAN V GPUs (each has 12GB device memory), and two Intel Xeon Gold 6140 CPU processors (each has 18 cores). The programs are compiled with CUDA-10.0 and GCC 7.3.0 with O3 flag.

6.2 Effects of Reuse

Figure 8 shows the evaluation results comparing our approach with the existing GPU methods. In the figure, the bars that reach the top mean the corresponding schemes have not completed after more than 3 hours.

Comparison with `GPSM`. Compared with `GPSM`, `RPS` achieves speedups of $1.2-3.0$, $2.5-4.8$, $1.3-3.3$, $1.6-4.1$, $1.8-5.0$, $1.6-2.8$ on the datasets YT, WK, TP, TW, DG, OR respectively. This confirms that reuse can accelerate

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3035564, IEEE Transactions on Knowledge and Data Engineering

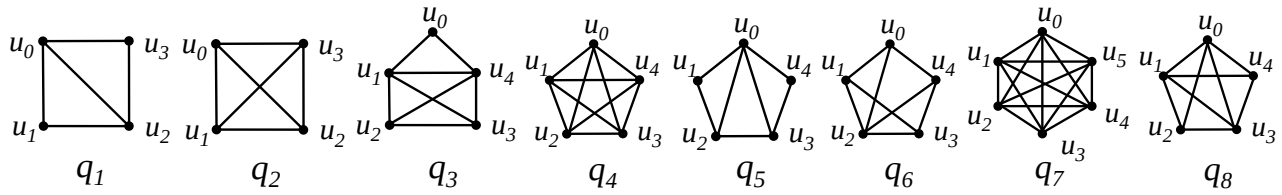


Fig. 7: Queries

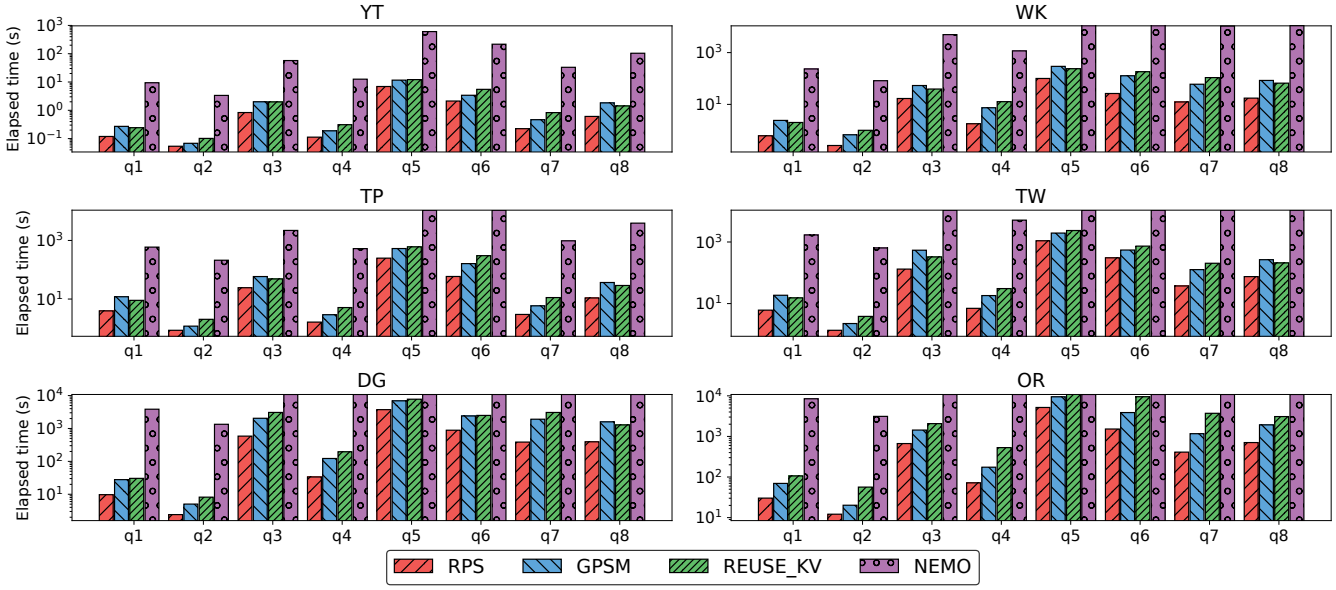


Fig. 8: Effects of reuse on GPUs.

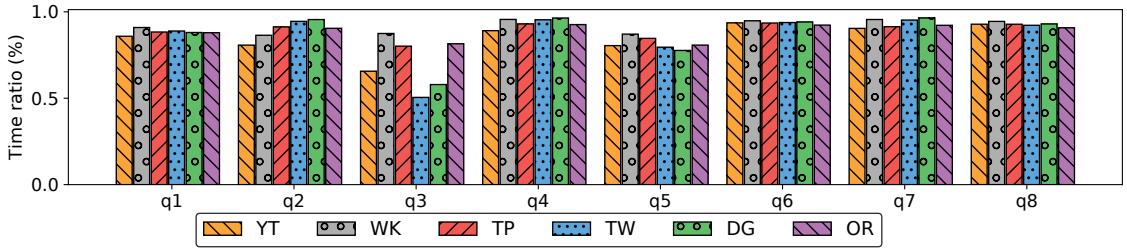


Fig. 9: The ratio of the execution time of set intersection operations versus the overall execution time. We run GPSM, the state-of-the-art GPU solutions, to profile the execution.

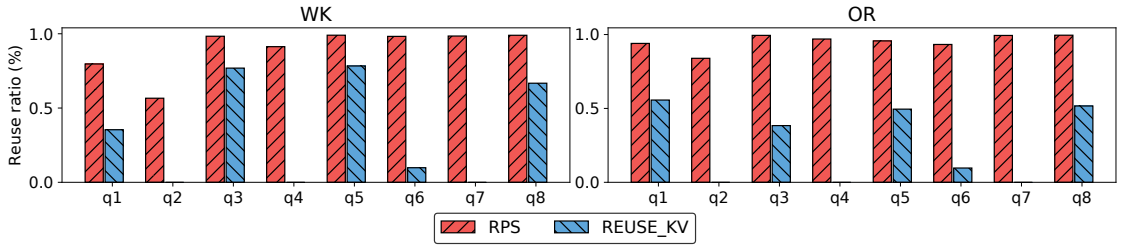


Fig. 10: The ratio of the set intersection operations that can be avoided by the reuse.

TABLE 3: Plan generation time (in milliseconds) and the ratio (100%) of plan generation versus total execution time of RPS.

Query	q1	q2	q3	q4	q5	q6	q7	q8
Time	0.13	0.17	0.96	1.81	0.58	0.78	48	1.22
WK	0.02	0.07	< 0.01	11	< 0.01	< 0.01	0.43	< 0.01
OR	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.01	< 0.01

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3035564, IEEE Transactions on Knowledge and Data Engineering

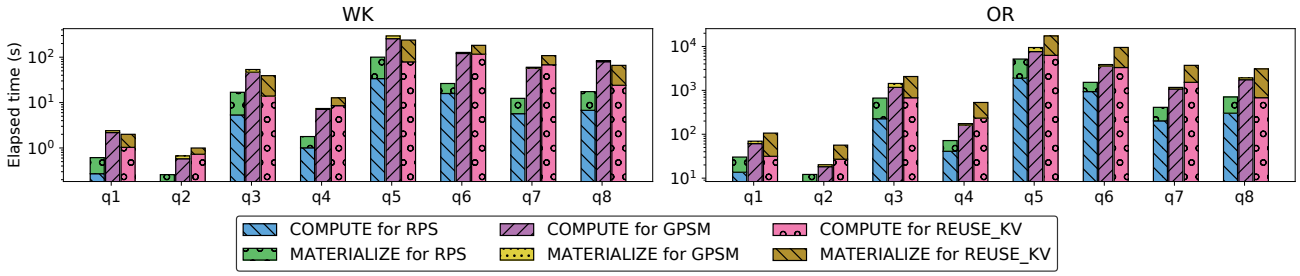


Fig. 11: The execution time breakdown of RPS, GPSM, and REUSE_KV.

runtime performance of parallel subgraph enumeration. The acceleration comes from the benefit of the smaller number of set intersection operations performed, since RPS reuses the cached results to avoid the redundant operations. This benefit reduces the workload executed in the COMPUTE procedure, and thus improves the overall performance. To verify that reuse reduces the number of set intersection operations and improves the COMPUTE procedure, we present the profiling results in the next subsection.

Comparison with REUSE_KV. REUSE_KV implements a hash table to track the cached results of set intersection operations. As the memory allocation on GPUs is slow, we optimize the implementation of REUSE_KV by batched memory allocation. Instead of allocating memory for each cached result, we allocate an array to store all the cached results in each iteration, and the hash table simply tracks the position of each cached result on the array. Even with this optimization, REUSE_KV still incurs large overheads in execution for concurrent hash table maintenance. The overheads can compromise and sometimes outweigh the benefit of reuse. Indeed, the performance of REUSE_KV is not better than that of GPSM, which is the implementation without the reuse. Our RPS always outperforms REUSE_KV because RPS exploits reuse efficiently and does not incur large overheads as in REUSE_KV.

Comparison with NEMO. Our RPS achieves significantly better performance than NEMO. Besides the reuse, another reason for this improvement is that NEMO uses an inefficient method to perform set intersection operations. To intersect the adjacent lists $N(v_1), N(v_2) \cdots N(v_n)$, NEMO chooses the adjacent list with the smallest cardinality, say $N(v_i)$, iterates each neighbor $v \in N(v_i)$, and then scans the adjacent list $N(v)$ to verify whether $N(v)$ contains the vertices $v_1, v_2 \cdots v_n$. This approach is shown to be efficient for small graphs [18] due to the coalesced memory access. However, it is rather inefficient for large graphs because the quadratic time complexity can significantly increase the workload for large adjacent lists.

6.3 Profile Results

Bottleneck analysis. The set intersection operation is the runtime bottleneck for parallel subgraph enumeration. We run the state-of-the-art GPU solution, GPSM, to profile the execution time of such an operation. As shown in Figure 9, the operation takes nearly 90% of the running time for most datasets and queries, and sometimes up to 95%.

Reuse ratio. We call the *reuse ratio* as the ratio of the redundant operations avoided by the reuse versus the total number of set intersection operations. We run RPS and REUSE_KV to profile the reuse ratios. As shown in Figure 10, RPS can avoid $42\% - 99\%$ of the set intersection operations. Even though GPU memory is limited, most cached results are still available for RPS. To perform the set intersection given the partial instance f, it will find another partial instance f^* corresponding to the cached result. In many cases, f and f^* share a common prefix and the pipeline processing will likely maintain both instances in the same pipeline. This leads to the high reuse ratio for RPS.

In comparison, the reuse ratio of REUSE_KV is much smaller than RPS. There are two reasons. First, REUSE_KV can only reuse the result generated by exactly the same set intersection operation, while RPS can even exploit the result from a different operation with the help of the reuse analysis framework. For example, for the clique $q2$, to compute the set intersection operation $N(v_1) \cap N(v_2) \cap N(v_3)$ among data vertices v_1, v_2, v_3, RPS can exploit the result $N(v_1) \cap N(v_2)$. However, REUSE_KV can reuse the result only if the same set intersection operation $N(v_1) \cap N(v_2) \cap N(v_3)$ was executed in the past, while such a condition is not true for q2. This makes the reuse ratio of REUSE_KV as low as 0 in cliques, i.e., q2, q4, q7.

Another reason of small reuse ratios is that our current implementation of REUSE_KV simply drops the hash table to release memory when its consumption exceeds the threshold. This can remove plenty of useful cached results and make them unavailable for later usage. One may think of selectively dropping a subset of data in the hash table to improve the reuse ratio but this is difficult to be implemented in GPUs: it is inefficient to track sophisticated metadata in a decentralized way, e.g., least recently used items, for cache eviction; removing a set of items concurrently from the hash table needs plenty of overheads [20]. Another consideration is to increase the memory threshold, which is set to 1GB currently. We think this amount is sufficiently large for reuse. A larger threshold cannot help because REUSE_KV can only salvage the results generated by exactly the same set intersection operations and a lot still cannot be reused. Moreover, a larger memory size for the hash table will increase overheads of GPU memory allocation/deallocation, which do not help to improve the overall performance.

Execution time breakdown. To investigate how reuse may improve performance, we run RPS, GPSM, and REUSE_KV to profile two major procedures in execution. As shown in Figure 11, RPS outperforms GPSM because the time on COMPUTE is reduced. In COMPUTE, RPS can exploit reuse and avoid a large number of set intersection operations but

GPSM has to perform such operations repeatedly. In MATERIALIZE, RPS spends more time than GPSM. To support reuse, RPS needs to maintain extra data structures, e.g., the tree structure to track the partial instances, which leads to additional costs compared with GPSM.

REUSE_KV achieves better performance than GPSM sometimes, but still cannot win over RPS because of two reasons. First, the reuse ratio of REUSE_KV is lower than RPS, as mentioned above. Thus, it needs to perform more set intersection operations and costs more time in COMPUTE. The second reason is that REUSE_KV spends a large amount of time in MATERIALIZE to concurrently insert set intersection results into hash table, which involves lots of irregular memory accesses and memory operations. This causes REUSE_KV to perform even worse than GPSM for some queries, i.e., the case without reuse.

6.4 Plan Generation

As shown in Table 3, the plan generation time takes less than 1 millisecond for most queries because the number of pattern vertices is small. This time is still rather small compared with the total execution time.

7 CONCLUSION

This paper studies reusing the execution results of set intersection operations to accelerate subgraph enumeration on GPUs. We propose the reuse analysis framework to generate the execution plan to utilize the cached results and the reusable parallel search to exploit the reuse in execution. The experimental evaluation shows that our approach outperforms the state-of-the-art GPU solutions and achieves the speedups of up to 5 times.

Acknowledgements. The project of this work is supported by the grant, MOE2017-T2-1-141, from Singapore Ministry of Education. Yuchen Li is partially supported by Lee Kong Chian fellowship.

REFERENCES

[1] N. Pržulj, D. G. Corneil, and I. Jurisica, "Efficient estimation of graphlet frequency distributions in protein–protein interaction networks," *Bioinformatics*, vol. 22, no. 8, pp. 974–980, 2006.

[2] J. A. Grochow and M. Kellis, "Network motif discovery using subgraph enumeration and symmetry-breaking," in *RECOMB*, vol. 4453. Springer, 2007, pp. 92–106.

[3] S. R. Kairam, D. J. Wang, and J. Leskovec, "The life and death of online groups: Predicting group growth and longevity," in *Proceedings of the fifth ACM international conference on Web search and data mining*. ACM, 2012, pp. 673–682.

[4] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.

[5] J. Leskovec, A. Singh, and J. Kleinberg, "Patterns of influence in a recommendation network," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2006, pp. 380–389.

[6] J. Ugander, L. Backstrom, and J. Kleinberg, "Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections," in *Proceedings of the 22nd international conference on World Wide Web*. ACM, 2013, pp. 1307–1318.

[7] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[8] W.-S. Han, J. Lee, and J.-H. Lee, "Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 337–348.

[9] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 617–628, 2015.

[10] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1199–1214.

[11] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 625–636.

[12] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *Proceedings of the VLDB Endowment*, vol. 8, no. 10, pp. 974–985, 2015.

[13] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 217–228, 2016.

[14] M. Qiao, H. Zhang, and H. Cheng, "Subgraph matching: on compression and computation," *Proceedings of the VLDB Endowment*, vol. 11, no. 2, pp. 176–188, 2017.

[15] B. Bhattarai, H. Liu, and H. H. Huang, "Ceci: Compact embedding cluster index for scalable subgraph matching," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD*, vol. 19, 2019.

[16] S. Sun and Q. Luo, "Efficient parallel subgraph enumeration on a single machine," in *2019 IEEE 35th International Conference on Data Engineering*, 2019.

[17] R. Raman, O. van Rest, S. Hong, Z. Wu, H. Chafi, and J. Banerjee, "Pgx. iso: parallel and efficient in-memory engine for subgraph isomorphism," in *Proceedings of Workshop on GRAph Data management Experiences and Systems*. ACM, 2014, pp. 1–6.

[18] W. Lin, X. Xiao, X. Xie, and X.-L. Li, "Network motif discovery: A gpu approach," in *ICDE*, 2015.

[19] H.-N. Tran, J.-j. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *International Conference on Database Systems for Advanced Applications*. Springer, 2015, pp. 299–315.

[20] S. Ashkiani, M. Farach-Colton, and J. D. Owens, "A dynamic hash table for the gpu," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 419–429.

[21] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen, "Horton tables: fast hash tables for in-memory data-intensive computing," in *2016 USENIX Annual Technical Conference USENIX ATC 16)*, 2016, pp. 281–294.

[22] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: a case for gpus to maximize the throughput of in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.

[23] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items," in *2015 USENIX Annual Technical Conference USENIX ATC 15)*, 2015, pp. 71–82.

[24] R. Gandhi, A. Gupta, A. Povzner, W. Belluomini, and T. Kaldewey, "Mercury: Bringing efficiency to key-value stores," in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013, p. 6.

[25] "Implementation of RPS," 2020. [Online]. Available: https://github.com/guowentian/SubgraphMatchGPU

[26] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 364–375, 2008.

[27] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[28] Y. Saad, "Sparskit: A basic tool kit for sparse matrix computations," 1990.

[29] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel triangle counting on gpus," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 171–182.

[30] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin, "Efficient parallel lists intersection and index compression algorithms using graphics processing units," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 470–481, 2011.

[31] J. Fox, O. Green, K. Gabert, X. An, and D. A. Bader, "Fast and adaptive list intersections on the gpu," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[32] L. Wang, Y. Wang, C. Yang, and J. D. Owens, "A comparative study on exact triangle counting algorithms on the gpu," in *Proceedings of the ACM Workshop on High Performance Graph Processing*. ACM, 2016, pp. 1–8.

[33] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[34] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han, "Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together," in *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019, pp. 1429–1446.

[35] S. Csun and Q. Luo, "Parallelizing recursive backtracking based subgraph matching on a single machine," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 1–9.

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

[37] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraph instances using map-reduce," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 2013, pp. 62–73.

[38] T. Plantenga, "Inexact subgraph isomorphism in mapreduce," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 164–175, 2013.

[39] D. Mawhirter and B. Wu, "Automine: harmonizing high-level abstraction and high performance for graph mining," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 509–523.

[40] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: an efficient and flexible graph mining system on cpu and gpu," *arXiv preprint arXiv:1911.06969*, 2019.

[41] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, "Gpu-accelerated subgraph enumeration on partitioned graphs," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1067–1082.

[42] W. Guo, Y. Li, M. Sha, and K.-L. Tan, "Parallel personalized pagerank on dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 11, no. 1, pp. 93–106, 2017.

[43] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on gpus," *Proceedings of the VLDB Endowment*, vol. 11, no. 1, pp. 107–120, 2017.

[44] M. Sha, Y. Li, and K.-L. Tan, "Gpu-based graph traversal on compressed graphs," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 775–792.

[45] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on gpus," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 147–156.

[46] B. Ding and A. C. König, "Fast set intersection in memory," *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 255–266, 2011.

[47] S. Kim, T. Lee, S.-w. Hwang, and S. Elnikety, "List intersection for web search: algorithms, cost models, and optimizations," *Proceedings of the VLDB Endowment*, vol. 12, no. 1, pp. 1–13, 2018.

[48] S. Han, L. Zou, and J. X. Yu, "Speeding up set intersections in graph algorithms using simd instructions," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1587–1602.

[49] B. Schlegel, T. Willhalm, and W. Lehner, "Fast sorted-set intersection using simd instructions." in *ADMS@ VLDB*, 2011, pp. 1–8.

[50] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: a gpu merging algorithm," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 331–340.

[51] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Building an efficient hash table on the gpu," in *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 39–53.

[52] "SNAP graph datasets," 2019. [Online]. Available: https://snap.stanford.edu/data/

[53] "Network repository," 2019. [Online]. Available: http://networkrepository.com

[54] X. Ren, J. Wang, W.-S. Han, and J. X. Yu, "Fast and robust distributed subgraph enumeration," *arXiv preprint arXiv:1901.07747*, 2019.

[55] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah, "Dualsim: Parallel subgraph enumeration in a massive graph on a single machine," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1231–1245.

Wentian Guo Wentian Guo received the BSc degree in computer science from South China University of Technology, Guangdong, China, in 2014 and the PhD degree in computer science from the School of Computing, National University of Singapore (NUS). His research aims to exploit modern hardwares to achieve fast and scalable data processing. He completed a PhD thesis on parallel graph processing with GPUs.

Yuchen Li Yuchen Li received the double BSc degrees in applied math and computer science (both degrees with first class honors) and the PhD degree in computer science from the National University of Singapore (NUS), in 2013 and 2016, respectively. He is an assistant professor with the School of Information Systems, Singapore Management University (SMU). Before joining SMU, he was a research fellow with the School of Computing, National University of Singapore. His research interests include graph analytics and heterogeneous computing.

Kian-Lee Tan Kian-Lee Tan received the PhD degree in computer science from the National University of Singapore (NUS), in 1994. He is a professor with the School of Computing, National University of Singapore. His current research interests include query processing and optimization in multiprocessor and distributed systems, database performance, data analytics, and database security. He was also a 2013 IEEE Technical Achievement Award recipient. He is an associate editor of the ACM Transactions on Database Systems (TODS) and the World Wide Web Journal. He has also served on the editorial boards of the Very Large Data Base (VLDB) Journal and the IEEE Transactions on Knowledge and Data Engineering (2009-2013). He is a member of the ACM and senior member of the IEEE