Fast Subgraph Matching on Large Graphs using Graphics Processors

Ha-Nguyen Tran(✉), Jung-jae Kim, and Bingsheng He

School of Computer Engineering, Nanyang Technological University, Singapore, Singapore
{s110035,jungjae.kim,bshe}@ntu.edu.sg

Abstract. Subgraph matching is the task of finding all matches of a query graph in a large data graph, which is known as an NP-complete problem. Many algorithms are proposed to solve this problem using CPUs. In recent years, Graphics Processing Units (GPUs) have been adopted to accelerate fundamental graph operations such as breadth-first search and shortest path, owing to their parallelism and high data throughput. The existing subgraph matching algorithms, however, face challenges in mapping backtracking problems to the GPU architectures. Moreover, the previous GPU-based graph algorithms are not designed to handle intermediate and final outputs. In this paper, we present a simple and GPU-friendly method for subgraph matching, called *GpSM*, which is designed for massively parallel architectures. We show that GpSM outperforms the state-of-the-art algorithms and efficiently answers subgraph queries on large graphs.

1 Introduction

Big networks from social media, bioinformatics and the World Wide Web can be essentially represented as graphs. As a consequence, common graph operations such as breadth-first search and subgraph matching face the challenging issues of scalability and efficiency, which have attracted increasing attention in recent years. In this paper, we focus on *subgraph matching*, the task of finding all *matches* or *embeddings* of a query graph in a large data graph. This problem has enjoyed widespread popularity in a variety of real-world applications, e.g., semantic querying [1,2], program analysis [3], and chemical compound search [4]. In such applications, subgraph matching is usually a bottleneck for the overall performance because it involves subgraph isomorphism which is known as an *NP-complete* problem [5].

Existing algorithms for subgraph matching are generally based on the *filtering-and-verification* framework [3,6–12]. First, they filter out all candidate vertices which cannot contribute to the final solutions. Then the verification phase follows, in which *backtracking*-based algorithms are applied to find results in an incremental fashion. Those algorithms, however, are designed to work only in small-graph settings. The number of candidates grows significantly high in medium-to-large-scale graphs, resulting in an exorbitant number of costly verification operations.

© Springer International Publishing Switzerland 2015
M. Renz et al. (Eds.): DASFAA 2015, Part I, LNCS 9049, pp. 299–315, 2015.
DOI: 10.1007/978-3-319-18120-2_18

Several indexing techniques have also been proposed for faster computation [3,9]; however, the enormous index size makes them impractical for large data graphs [14]. Distributed computing methods [13,14] have been introduced to deal with large graphs by utilizing parallelism, yet there remains the open problem of high communication costs between the participating machines.

Recently, GPUs with massively parallel processing architectures have been successfully leveraged for fundamental graph operations on large graphs, including breadth-first search [15,16], shortest path [15,17] and minimum spanning tree [18]. Traditional backtracking approaches for subgraph matching, however, cannot efficiently be adapted to GPUs due to two problems. First, GPU operations are based on *warps* (which are groups of threads to be executed in single-instruction-multiple-data fashion), and different execution paths generated by backtracking algorithms may cause a so-called *warp divergence* problem. Second, GPU implementations for coalesced memory accesses are no longer straightforward due to irregular access patterns [19].

To address these issues, we propose an efficient and scalable method called *GpSM*. GpSM runs on GPUs and takes on edges as the basic unit. Unlike previous *backtracking*-based algorithms, GpSM joins candidate edges in parallel to form partial solutions during the verification phase, and this procedure is conducted repeatedly until the final solution is obtained. An issue raised by such parallel algorithms is the considerable amount of intermediate results for joining operations, while backtracking algorithms only need to store less of such data during execution. We resolve this issue by adopting the pruning technique of [20], further enhancing it by ignoring *low-connectivity vertices* which have little or no effect of decreasing intermediate results during filtering.

To highlight the efficiency of our solution, we perform an extensive evaluation of GpSM against state-of-the-art subgraph matching algorithms. Experiment results on both real and synthetic data show that our solution outperforms the existing methods on large graphs.

The rest of the paper is structured as follows. Section 2 gives formal definitions and related works. In section 3, we introduce the filtering-and-joining approach to solve the problem. The filtering and joining phases are discussed in Section 4 and 5. Section 6 extends our method to deal with large graphs. Experiment results are shown in Section 7. Finally, Section 8 concludes our paper.

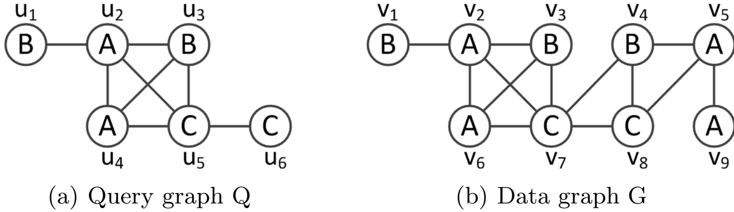
2 Preliminaries

2.1 Subgraph Matching Problem

We give a formal problem statement using *undirected labeled graphs*, though our method can be applied to *directed labeled graphs* as shown in the Experiment Results section.

Definition 1. *A* **labeled graph** *is a 4-tuple* $G = (V, E, L, l)$*, where* V *is the set of vertices,* $E \subseteq V \times V$ *is the set of edges,* L *is the set of labels and* l *is a labeling function that maps each vertex to a label in* L*.*

Definition 2. *A graph* $G = (V, E, L, l)$ *is* **subgraph isomorphic** *to another graph* $G' = (V', E', L', l')$*, denoted as* $G \subseteq G'$*, if there is an injective function (or a* **match***)* $f : V \rightarrow V'$*, such that* $\forall (u, v) \in E$*,* $(f(u), f(v)) \in E'$*,* $l(u) = l'(f(u))$*, and* $l(v) = l'(f(v))$*.*



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

Fig. 1. Sample query and data graph

Subgraph Matching Problem is defined as follows: Given a large data graph G and a query graph Q, we find all matches of Q in G. For example, the subgraph matching solution of the query graph Q in the data graph G in Figure 1 is $\{(u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_6), (u_5, v_7), (u_6, v_8)\}$.

Definition 3. *Given a query graph* $Q = (V, E, L, l)$ *and a data graph* $G = (V', E', L', l')$*, a vertex* $v \in V'$ *is called a* **candidate** *of a vertex* $u \in V$ *if* $l(u) = l'(v)$*,* $degree(u) \leq degree(v)$ *where* $degree(u)$*,* $degree(v)$ *are the number of vertices connected to edges starting vertex* u *and* v *respectively. The set of candidates of* u *is called* **candidate set** *of* u*, denoted as* $C(u)$*.*

The query vertex u_3 in Figure 1a has a label of B and a degree of 3. For the data graph vertex v_3 in Figure 1b, the label is also B and the degree is 3 which is equal to the degree of u_3. Therefore, v_3 is a candidate of u_3. The candidate set of u_3 is C(u_3) = $\{v_3, v_4\}$.

An *adjacency list* of a vertex u in a graph G is a set of vertices which are the destinations of edges starting from u, denoted as *adj(u)*. For example, the adjacency list of u_3 is $adj(u_3) = \{u_2, u_4, u_5\}$.

2.2 Subgraph Matching Algorithms

Most of state-of-the-art subgraph matching algorithms are based on backtracking strategies which find matches by either forming partial solutions incrementally or pruning them if they cannot produce the final results, as discussed in the works of Ullman [6,10], VF2 [7], QuickSI [8], GADDI [9], GraphQL [1] and SPath [3]. One of open issues in those methods is the selection of matching order (or visit order). To address this issue, TurboISO [11] introduces the strategies of candidate region exploration and combine-and-permute to compute a 'good' visit order, which makes the matching process efficient and robust.

To deal with large graphs, Sun et al. [14] introduce a parallel and distributed algorithm (which we call STW in this paper), in which they decompose the query graphs into 2-level trees, and apply graph exploration and join strategy to obtain

solutions in a parallel manner over a distributed memory cloud. Unlike STW, our method uses GPUs in order to keep the advantages of parallelism during computation, while simultaneously avoiding high communication costs between participating machines.

2.3 General-Purpose Computing on GPUs

GPUs are widely used as commodity components in modern-day machines. A GPU consists of many individual multiprocessors (or *SMs*), each of which executes in parallel with the others. During runtime, threads on each multiprocessor are organized into thread blocks, and each block consists of multiple 32-thread groups, called *warps*. If threads within a warp are set to execute different instructions, they are called *diverged*; computations in diverged threads are only partially parallel, thus reducing the overall performance significantly. The GPU includes a large amount of device memory with high bandwidth and high access latency, called *global memory*. In addition, there is a small amount of *shared memory* on each multiprocessor, which is essentially a low latency, high bandwidth memory running at register speeds.

Due to such massive amounts of parallelism, GPUs have been adopted to accelerate data and graph processing [15,16,23,27]. Harish et al. [15] implement several common graph algorithms on GPUs including BFS, single source shortest path and all-pair shortest path. Hong et al. [23] enhance BFS by proposing a virtual warp-centric method to address the irregularity of BFS workload. Merrill et al. [16] propose a BFS algorithm which is based on fine-grained task management and built upon an efficient prefix sum; this work has generally been considered as one of the most complete and advanced works regarding BFS traversal on GPUs. Finally, Medusa is a general programming framework for graph processing in GPU settings [25], providing a rich set of APIs based on which developers can further build their applications.

3 GpSM Overview

We introduce a GPU-based algorithm called GpSM to solve the problem of subgraph matching. Unlike previous CPU methods with complicated pruning and processing techniques, our algorithm is *simple* and *designed for massively parallel architectures*.

3.1 Filtering-and-Joining Approach

We find that STW method [14] simultaneously filters out candidate vertices and matches of basic units (i.e. 2-level trees), and thus generates a large amount of irrelevant candidate vertices and edges. Our method performs the two tasks separately in order to reduce intermediate results. The main routine of the GPU-based algorithm is illustrated in Algorithm 1.

The inputs of GpSM are a connected query graph q and a data graph g. The vertex sets and edge sets of q and g are V_q, E_q and V_g, E_g respectively. The output is a set of subgraph isomorphisms (or matches) of q in g. In our method, we present a match as a list of pairs of a query vertex and its mapped data vertex. Our solution is the collection M of such lists.

Algorithm 1. GpSM(q,g)

Input: query graph q, data graph g
Output: all matches of q in g

1 C := `InitializeCandidateVertices(`q,g`);`
2 C := `RefineCandidateVertices(`q,g,C`);`
3 E := `FindCandidateEdges(`q,g,C`);`
4 M := `JoinCandidateEdges(`q,g,E`);`
5 **return** M

Our method uses a *filtering-and-joining* strategy. The filtering phase consists of two tasks. The first task filters out candidate vertices which cannot be matched to query vertices (Line 1). After this task there can still be a large set of irrelevant candidate vertices which cannot contribute to subgraph matching solutions. The second task continues pruning this collection by calling a refining function, *RefineCandidateVertices*. In the function, candidate sets of query vertices are recursively refined either until no more candidates can be pruned, or up to a predefined number of times (Line 2). The details of the filtering phase will be discussed in Section 4. In the joining phase, GpSM collects candidate edges based on the candidate vertices obtained in the previous phase (Line 3) and combines them to produce the final subgraph matching solutions (Line 4) which are finally returned to users. Section 5 gives the detailed implementation of the joining phase.

3.2 Graph Representation

In order to support graph query answering on GPUs, we use three arrays to represent a graph $G = (V, E)$: *vertices array*, *edges array*, and *labels array*. The edges array stores the adjacency lists of all vertices in V, from the first vertex to the last one. The vertices array stores the start indexes of the adjacency lists, where the i-th element of the vertices array has the start index of the adjacency list of the i-th vertex in V. The labels array maintains labels of vertices in order to support our method on labelled graphs. The first two arrays have been used in previous GPU-based algorithms [15,16,23]. Figure 2 shows the representation of the graph illustrated in Figure 1a in the GPU memory.

The advantage of the data structure is that vertices in the adjacency list of a vertex are stored next to each other in the GPU memory. During GPU execution, consecutive threads can access consecutive elements in the memory. Therefore, we can avoid the random access problem and decrease the accessing time for GPU-based methods consequently.

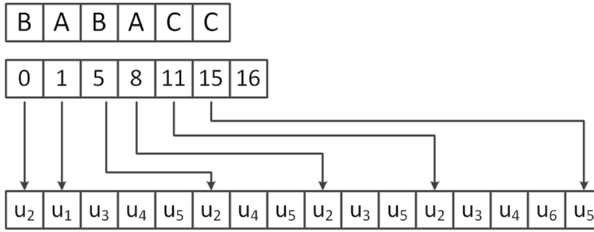


Fig. 2. Graph Representation in GPU Memory

4 Filtering Phase

This section describes the implementation of the filtering phase on GPUs. The purpose of this phase is to reduce the number of candidate vertices and thus decrease the amount of candidate edges as well as the running time of the joining phase. The filtering phase consists of two tasks: initializing candidate vertices and refining candidate vertices.

4.1 Initializing Candidate Vertices

The first step of the filtering phase is to initialize candidate sets of all query vertices. In the task, we take a spanning tree generated from the query graph as the input. This section presents a heuristic approach to selecting a good spanning tree among many spanning trees of the query graph. The approach is based on the observation that if the filtering starts from the query vertices with the smallest number of candidates, its intermediate results can be kept to the minimum. Since we do not know the number of candidates in the beginning, we estimate it by using a vertex ranking function $f(u) = \frac{deg(u)}{freq(u.label)}$ [11,14], where *deg(u)* is the degree of u and *freq(u.label)* is the number of data vertices having the same label as u.

We find a spanning tree T and a visit order O for a query graph as follows: Initially, we pick a query edge (u, v) such that $f(u) \geq f(v)$ and $f(u) + f(v)$ is the maximum among all query edges. We add u to the visit order O, and add the edges connected to u to the spanning tree T, except those whose endpoints are already in the vertices set of T, i.e. $V(T)$. The process continues to pick up another query edge connected to T and add to O and T until no edge remains. Figure 5a depicts the spanning tree of the Figure 1a graph. Also, the visit order is u_5, u_2.

Algorithm 2 outlines the task of finding candidate vertices of each query vertex from the data graph, following the visit order obtained earlier. For each query vertex u, GpSM first checks if each of data vertex is a candidate of u and keeps the candidacy information in the Boolean array $c_set[u]$ in parallel ($kernel_check$[1]; Line 7) in the case that its candidate set is not initialized (Line

[1] Note that all functions whose names start with *kernel* are device functions that run on GPUs.

6). It then creates an integer array (*c_array*) that collects the indexes of candidates of u from $c_set[u]$ (*kernel_collect*; Line 9). GpSM calls another device function (*kernel_explore*; Line 10) that prunes out all candidate vertices u' of u such that there is a vertex $v \in adj(u)$ which has no candidate vertex in $adj(u')$ (Lines 16-18), and explores the adjacency list of u in the spanning tree in order to filter the candidates of the vertices in $adj(u)$ (Lines 19-22). Thus, the final outputs are *Boolean* arrays *c_set*, which represent the filtered candidate sets of query vertices.

Algorithm 2. Initializing candidate vertices

```
   Input: spanning tree T, data graph g
   Output: candidate sets of vertices c_set
 1 Algorithm InitializeCandidateVertices(T, g)
 2     foreach vertex u ∈ T do
 3         c_set[u][v] := false; ∀v ∈ V_g
 4         initialized[u] := false;
 5     foreach u ∈ T in the visit order do
 6         if initialized[u] = false then
 7             kernel_check(c_set[u], g);
 8             initialized[u] := true;
 9         c_array := kernel_collect(u, c_set[u]);
10         kernel_explore(u, c_array, c_set, T, g);
11         foreach v ∈ adj(u) do
12             initialized[v] := true;
13     return c_set;
14 Procedure kernel_explore(u, c_array, c_set, T, g)
15     u' := GetCandidate(c_array, warp_id);
16     if exist v ∈ adj(u) such that no v' ∈ adj(u') is a candidate of v then
17         c_set[u][u'] := false;
18         return;
19     foreach v ∈ adj(u) do
20         v' := GetAdjacentVertex (u', thread_id);
21         if v' is a candidate of v then
22             c_set[v][v'] := true;
```

GPU Implementation: We implement the two GPU device functions *kernel_collect* and *kernel_explore* in the first step of the filtering phase, based on two optimization techniques: *occupancy maximization* to hide memory access latency and *warp-based execution* to take advantage of the coalesced access and to deal with workload imbalance between threads within a warp. We skip details of the device function *kernel_check* since its implementation is straightforward.

1) *kernel_collect*. This function is to maximize the occupancy of the *kernel_explore* execution. At runtime, warps currently running in an SM are called

active warps. Due to the resource constraints, each SM allows a maximum number of active warps running concurrently at a time. Occupancy is the number of concurrently running warps divided by the maximum number of active warps. At runtime, when a warp stalls on a memory access operation, the SM switches to another active warp for arithmetic operations. Therefore, high-occupancy SM is able to adequately hide access latency.

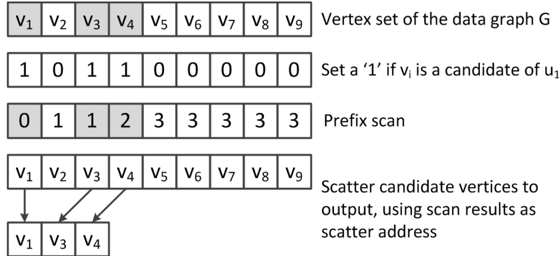


Fig. 3. Collect candidate vertices of u_1

A naive approach to executing *kernel_explore* is that only the warps corresponding to the *true* elements of $c_set[u]$ continue filtering vertices in $adj(u)$. However, the approach suffers from the low-occupancy problem since warps with the $false$ elements are idle. For example, we assume that the maximum number of active warps on the multiprocessor is 3. In the first 3 active warps, the occupancy is 66.66% because only the warps corresponding to v_1 and v_3 execute *kernel_explore* while the warp with v_2 is idle. For the next 3 concurrently running warps, the occupancy is only 33.33%. GpSM resolves the issue by adopting a stream compaction algorithm [26] to gather candidate vertices into an array *c_array* for those $c_set[u]$ with true values. The algorithm employs prefix scan to calculate the output addresses and to support writing the results in parallel. The example of collecting candidate vertices of u_1 is depicted in Figure 3. By taking advantage of *c_array*, all 3 active warps are used to explore the adjacency lists of v_1, v_3 and v_4. As a result, our method achieves a high occupancy.

2) *kernel_explore*. Inspired by the warp-based methods used in BFS algorithms for GPUs [16,23], we assign to each warp a candidate vertex $u' \in C(u)$ (or *c_array* from *kernel_collect*). Within the warp, consecutive threads find the candidates of $v \in adj(u)$ in $adj(u')$. This method takes advantage of coalesced access since the vertices of $adj(u')$ are stored next to each other in memory. It also addresses the warp divergence problem since threads within the warp execute similar operations. Thus, our method efficiently deals with the workload imbalance problem between threads in a warp. Figure 4 shows an example of filtering candidate vertices of u_2 based on the candidate set of u_1, $C(u_1) = \{v_1, v_3, v_4\}$.

If a data vertex has an exceptionally large degree compared to the others, GpSM deals with it by using an entire block instead of a warp. This solution reduces the workload imbalance between warps within the block.

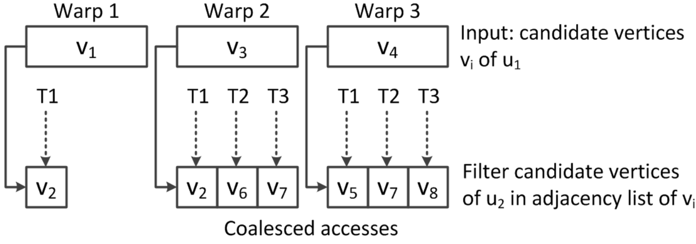
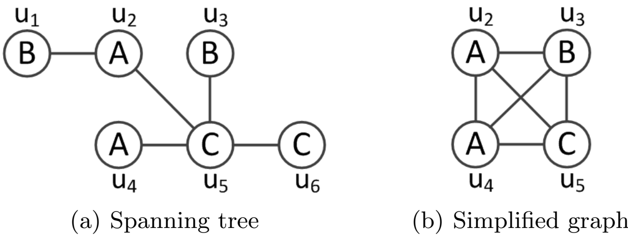


Fig. 4. Filter candidate vertices of u_2 based on adjacency lists of $C(u_1) = \{v_1, v_3, v_4\}$

4.2 Refining Candidate Vertices

After filtering out candidate vertices for the first time, there can be still a large number of candidate vertices which cannot be parts of final solutions. To address this issue, we propose a recursive filtering strategy to further prune irrelevant candidate vertices. The size of candidate edges and intermediate results are then reduced consequently.

We observe the followings: 1) Exploring non-tree edges (i.e. those that form cycles) can reduce the number of irrelevant candidates significantly; and 2) the more edges a vertex has, the more irrelevant candidates of the vertex the filtering techniques aforementioned can filter out. Based on the first observation, from the second round of the filtering process, our method uses the original query graph for exploration rather than a spanning tree of the query graph. Based on the second observation, our method ignores query vertices connected to small number of edges, called *low connectivity vertices*. For small-size query graphs, a low connectivity vertex has the degree of 1. As for big query graphs, we can increase the value of degree threshold to ignore more low connectivity vertices. The query graph obtained after removing low connectivity vertices from Q is shown in Figure 5b.



(a) Spanning tree (b) Simplified graph

Fig. 5. Spanning tree and simplified graph of Q

GPU implementation: The main routine of the refining task is similar to the filtering in the previous section. The differences are as follows: 1) *kernel_check* is not necessary for the refining process and 2) we only use the pruning task (Lines 16-18) in the *kernel_explore* function. By taking advantage of the *c_set* array

generated in the initialization step, the refinement can verify the candidate conditions easily and reduce the random accesses during the candidate verification.

Ideally, the optimal candidate sets of query vertices are obtained when the refinement is recursively invoked until no candidate is removed from the candidate sets. However, our experiments show that most of irrelevant candidates are pruned in the first few rounds. The later rounds do not prune out many candidates, but lead to inefficiency and reduce the overall performance. Therefore, the refining task terminates after a limited number of rounds.

In the tasks of initializing and refining candidate sets of query vertices, GpSM requires $O(|V_q| \times |V_g|)$ space to maintain *Boolean* arrays which are used to collect candidate vertices and $O(|V_g|)$ space to keep the collected set. Let S be the number of SMs. Each SM has P active threads. For each visited vertex, the prefix scan in *kernel_collect* executes in $O(|V_g| \times log(|V_g|)/(S \times P))$ time while *kernel_explore* runs in $O(|V_g| \times |d_g|/(S \times P))$, where d_g is the average degree of the data graph. Assume that the candidate refinement stops after k rounds, the total time complexity of the filtering phase is $O(|V_q| \times k \times (|V_g| \times log(|V_g|) + |V_g| \times |d_g|)/(S \times P))$.

5 Joining Phase

In the joining phrase, GpSM first gathers candidate edges in the data graph and then combines them into subgraph matching solutions.

The output of each query edge (u, v) in the task of gathering candidate edges is represented as a hash table, as depicted in Figure 6. The keys of this table are candidate vertices u' of u, and the value of a key u' is the address of the first element of the collection of candidate vertices v' of v such that $(u', v') \in E_g$. An issue of the step is that the number of the candidate edges is unknown, and thus that we cannot directly generate such a hash table. To address this issue, we employ the *two-step output scheme* [22] as follows: 1) Given a query edge (u, v), each warp is assigned to process a candidate vertex u' of u and counts the number of candidate edges starting with u' (designated as (u', v')). The system then computes the address of the first v' for u' in the hash table of (u, v). 2) It then re-examines the candidate edges and writes them to the corresponding addresses of the hash table.

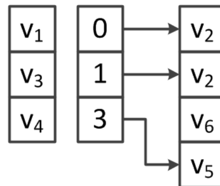


Fig. 6. Candidate edges of (u_1, u_2)

After finding the candidate edges, GpSM combines them to produce subgraph matching solutions as follows: Initially, we pick a query edge (u, v) with the

smallest number of candidate edges, and mark as *visited* the vertices u, v and the edge (u, v). Here the candidates of (u, v) are partial subgraph matching solutions. We select the next edge among the unvisited edges of the query graph, denoted by (u', v'), such that 1) both u' and v' are visited vertices, or 2) if there is no such edge, either u' or v' is a visited vertex. If there are multiple such edges, we select the one with the smallest number of candidates. Candidate edges of (u', v') are then combined with the partial solutions. The procedure is conducted repeatedly until all query edges are visited.

GPU Implementation: The GPU implementation for the task of gathering candidate edges is similar to that of the filtering phase, except introducing the two-step output scheme. For the task of combining partial subgraph matching solutions, we apply the warp-based approach as follows: Each warp i is responsible for combining a partial solution $M_i(q)$ with candidate edges of (u, v), where u is already visited. First, the warp retrieves the candidate vertex of u from $M_i(q)$ (e.g., u'). It looks up the hash table storing candidate edges of (u, v) to find the key u' and retrieve the candidate vertices v' of v from the hash table. By using our data structure of candidate edges, this task can be done in logarithmic time. Threads within the warp then verify whether (u', v') can be merged to $M_i(q)$, in which GpSM again follows the two-step output scheme to write the merged results.

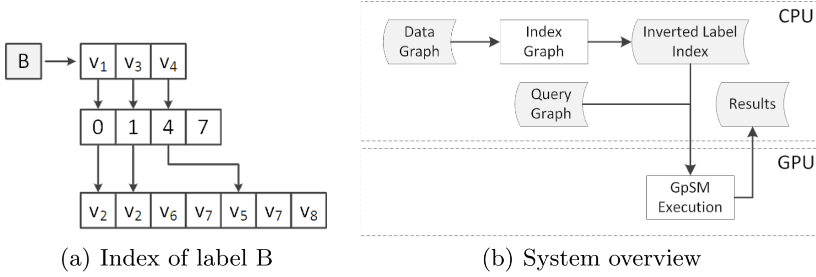
Shared Memory Utilization. The threads within the warp i should share the partial solution $M_i(q)$ and access them frequently. We thus store and maintain $M_i(q)$ in the shared memory instead of the device memory, which efficiently hides the memory stalls.

Let $C(e_i)$ be the candidate edges of the edge e_i. The joining phase is done in $O(\prod_{i=1}^{|E_q|} |C(e_i)| \times log(|V_g|)/(S \times P)$ time. Note that the running time of the joining phase highly depends on the number of candidates of query edges. Therefore, reducing the number of candidate vertices in the filtering phase plays an important role in decreasing both the running time and the memory used to maintain partial solutions.

6 Extended GpSM for Very Large Graphs

In real-world applications, the sizes of data graphs might be too large to be stored in the memory of a single GPU device. In general, such large graphs have many labels, and thus the vertices corresponding to the labels of a given query graph, together with their adjacency lists, are relatively small. Based on the observation, we make an assumption that the relevant data of query graph labels are small enough to fit into the GPU memory. Therefore, we can make GpSM work efficiently on large graphs by storing them with the *inverted-vertex-label* index data structure in CPU memory or hard disk and, given a query graph, by retrieving only relevant vertices and their adjacency lists to the GPU memory.

For each label l in the data graph G, we use three array structures. The first array contains all vertices that have the label of l (designated as V_l.) The other



(a) Index of label B (b) System overview

Fig. 7. GpSM Solution on large graphs

two arrays are the vertices and edges arrays corresponding to V_l, as defined in Section 3.2. An entry of the inverted-vertex-label index for label B of the data graph in Figure 1b is depicted in Figure 7a.

Figure 7b provides a system overview of our solution for large-graph subgraph matching using both GPUs and CPUs. Here rectangles indicate tasks while the others represent data structures used in our solution. The first task is to create an inverted-vertex-label index which is then stored in the hard disk. In order to decrease the time to transfer data from the hard disk, we keep the most frequent vertex labels in the main memory. Given a query graph, our solution retrieves all the data associated with the query labels from the main memory or hard disk and transfers them to the GPU memory to further search for subgraph matching solutions, which are finally returned to the main memory.

7 Experiment Results

We evaluate the performance of GpSM in comparison with state-of-the-art subgraph matching algorithms, including VF2 [7], QuickSI (in short, QSI) [8], GraphQL (in short, GQL) [1] and TuroboISO [11]. The implementations of VF2, QuickSI and GraphQL used in our experiments are published by Lee and colleages [21]. As for TurboISO, we use an executable version provided by the authors.

Datasets. The experiments are conducted on both real and synthetic datasets. The real-world data include the Enron email communication network and the Gowalla location-based social network[2]. On the other hand, the synthetic datasets are generated by RMAT generator [24], and vertices are labeled randomly. As for query graphs, given the number of vertices N and the average number of edges per vertex D (called *degree*), we generate connected labeled query graphs of size N, randomly connecting the vertices and making their average degree D. Except for experiments with varying degrees, the query graphs always have the degree of 2.

Environment. The runtime of the CPU-based algorithms is measured using an Intel Core i7-870 2.93 GHz CPU with 8GB of memory. Our GPU algorithms

[2] These datasets can be downloaded from Stanford Dataset Collection website. See https://snap.stanford.edu/data for more details.

are tested using CUDA Toolkit 6.0 running on the NVIDIA Tesla C2050 GPU with 3 GB global memory and 48 KB shared memory per Stream Multiprocessor. For each of those tests, we execute 100 different queries and record the average elapsed time. In all experiments, algorithms terminate only when all subgraph matching solutions are found.

7.1 Comparison with State-of-the-art CPU Algorithms

The first set of experiments is to evaluate the performance of GpSM compared to the state-of-that-art algorithms. These comparisons are performed on both synthetic and real datasets. The input graphs are *undirected graphs* because the released version of TurboISO only works with undirected graphs.

Synthetic Datasets. The size of the synthetic data graphs of the first experiment set varies from 10,000 vertices to 100,000 vertices. All the data graphs have 10 distinct labels and the average degree of 16, and can fit into GPU memory. The query graphs contain 6 vertices and 12 edges. Figure 8 shows that GpSM clearly outperforms VF2, QuickSI and GraphQL. Compared to TurboISO, our GPU-based algorithm runs slightly slower when the size of the data graphs is relatively small (i.e. 10,000 vertices). However, when the size of data graphs increases, GpSM is more efficient than TurboISO. We thus make further comparisons with TurboISO in more experiment settings.

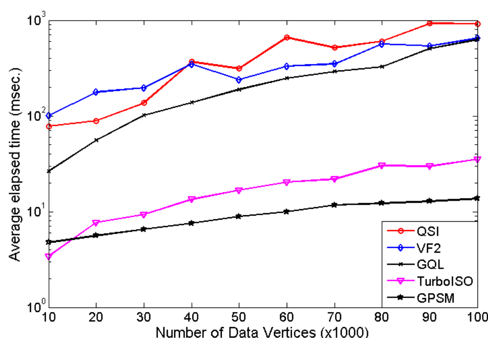
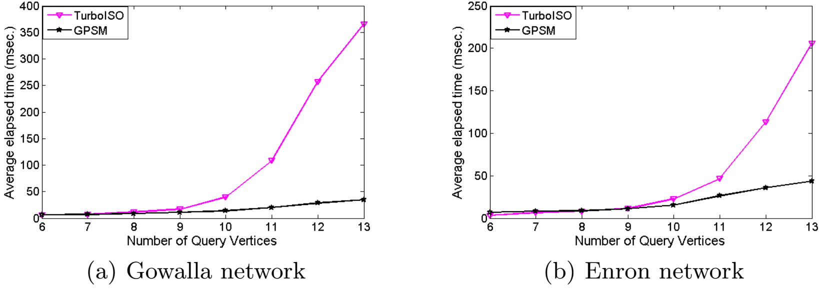


Fig. 8. Varying data sizes

Real Datasets. As for real-world datasets, Gowalla network consists of 196,591 vertices and 950,327 edges while Enron network has 36,692 vertices and 183,831 edges. In these experiments, we use 20 labels for Gowalla nerwork and 10 labels for Enron network. The number of query vertices varies from 6 to 13.

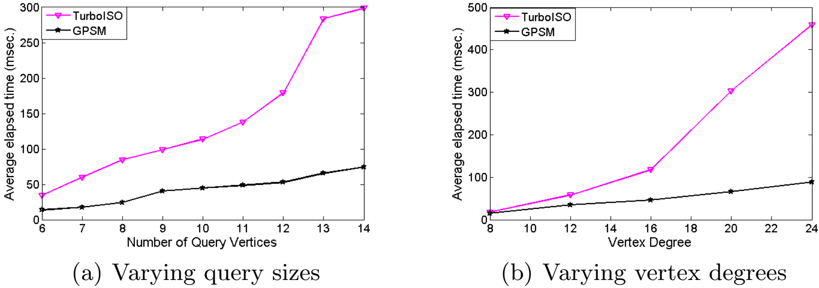
Figure 9a shows that TurboISO anwsers the subgraph matching queries against the Gowalla network efficiently when the size of query graphs is small. As the number of vertices increases, however, the processing time of TurboISO



(a) Gowalla network (b) Enron network

Fig. 9. Experiment on real datasets

grows exponentially. In contrast, GpSM shows almost linear growth. The two methods show similar performance difference when evaluated against the Enron network, as plotted in Figure 9b.



(a) Varying query sizes (b) Varying vertex degrees

Fig. 10. Comparison with TurboISO

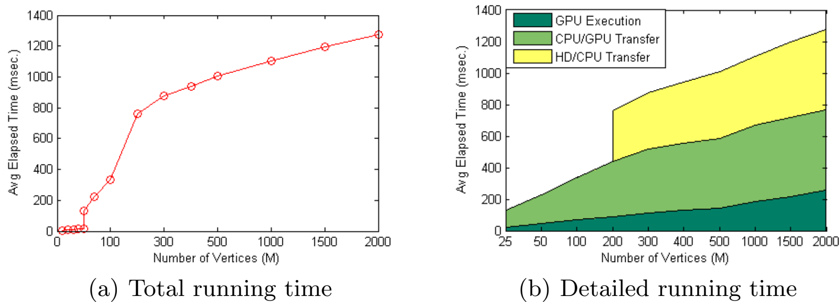
Comparison with TurboISO. We also compare the performance of GpSM with TurboISO, varying the size of query graphs and the degree of the data graphs, as shown in Figure 10. The data graphs are synthetic undirected graphs with 100,000 vertices and 10 labels.

Figure 10a shows the performance results of GpSM and TurboISO on the query graphs whose numbers of vertices vary from 6 to 14. In the experiment, the degree of the data graph is 16. Figure 10b shows their performance results when the vertex degree increases from 8 to 24, where the query graph size is fixed to 10. As shown in the two figures, the performance of TurboISO drops significantly while that of GpSM does not. This may be due to the fact that the number of recursive calls of TurboISO grows exponentially with respect to the size of query graphs and the degree of the data graph. In contrast, GpSM takes advantage of the large number of threads in GPUs to handle candidate edges in parallel and thus keep the processing time rising slowly.

7.2 Scalability Tests

We test the extended GpSM against very large graphs. The data graphs are *directed graphs* which are generated using the RMAT generator. The number of data vertices varies from 1 million to 2 billion vertices while the number of labels varies from 100 to 2000 according to the vertex number. The average degree of the data vertices is 8. The data graph is stored as follows: When the data graph is small, i.e from 1 million to 25 million vertices, we store it in the GPU global memory. If the vertex number of the data graph is between 25 million and 100 million, CPU memory is used to maintain the data graph. For data graphs with 200 million vertices and above, we store them in both CPU memory and hard disk. The largest number of vertices per label of the 25-million-vertex graph is around 350,000 while that of the 2-billion-vertex graph is nearly 1,400,000. The query graphs used in the experiments consist of 10 vertices and 20 edges.

When the data graph size is 25 million vertices, we perform two experiments. The first one maintains the whole data graph in GPU memory and the second uses CPU memory. As shown in Figure 11a, the second experiment answers subgraph matching queries slower than the first one, due to the time for data transfer from CPU memory to GPU memory.



(a) Total running time (b) Detailed running time

Fig. 11. Scalability Tests

The details of the running time are shown in Figure 11b, from which we observe the followings: 1) The time taken for GPU execution (i.e. subgraph matching) grows linearly as the data graph size increases, as expected from our time complexity analysis in Sections 4 and 5. 2) The GPU execution time takes around 11∼20% of the total running time, while the rest is taken by data transfers between GPU and CPU or hard disk. 3) The data transfer time also grows almost linearly as the data graph size increases, though the transfer from hard disk adds additional running time.

8 Conclusions

In this paper, we introduce an efficient method which takes advantage of GPU parallelism to deal with large-scale subgraph matching problem. Our method

called GpSM is simple and designed for massively parallel processing. GpSM is based on *filtering-and-joining* approaches, efficient GPU techniques of coalescence, warp-based and shared memory utilization, and a recursive refinement function for pruning irrelevant candidate vertices. Experiment results show that our method outperforms previous backtracking-based algorithms on CPUs and can efficiently answer subgraph matching queries on large graphs. In future, we will further improve the efficiency of GpSM for large graphs, for example, by dealing with large amount of intermediate results that do not fit into the GPU memory and also by adopting buffering techniques.

Acknowledgments. We thank to Prof. Wook-Shin Han and Dr. Jeong-Hoon Lee for sharing *iGraph* source code and executable files of $Turbo_{ISO}$ algorithm and providing clear explanations about $Turbo_{ISO}$. Bingsheng He is partly supported by a MoE. AcRF Tier 2 grant (MOE2012-T2-2-067) in Singapore.

References

1. He, H., Singh, A.K.: Graphs-at-a-time: query language and access methods for graph databases. In: SIGMOD, pp. 405–418 (2008)
2. Kasneci, G., Suchanek, F.M., Ifrim, G., Ramanath, M., Weikum, G.: Naga: Searching and ranking knowledge. In: ICDE, pp. 953–962 (2008)
3. Zhao, P., Han, J.: On graph query optimization in large networks. PVLDB **3**(1–2), 340–351 (2010)
4. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: SIGMOD, pp. 335–346 (2004)
5. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC, pp. 151–158 (1971)
6. Ullmann, J.R.: An algorithm for subgraph isomorphism. JACM **23**(1), 31–42 (1976)
7. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. PAMI **26**(10), 1367–1372 (2004)
8. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. PVLDB **1**(1), 364–375 (2008)
9. Zhang, S., Li, S., Yang, J.: GADDI: distance index based subgraph matching in biological networks. In: EDBT, pp. 192–203 (2009)
10. Ullmann, J.R.: Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. JEA **15**, 1–6 (2010)
11. Han, W.S., Lee, J., Lee, J.H.: Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: SIGMOD, pp. 337–348 (2013)
12. Kim, S., Song, I., Lee, Y.J.: An edge-based framework for fast subgraph matching in a large graph. In: Yu, J.X., Kim, M.H., Unland, R. (eds.) DASFAA 2011, Part I. LNCS, vol. 6587, pp. 404–417. Springer, Heidelberg (2011)
13. Brocheler, M., Pugliese, A., Subrahmanian, V.S.: COSI: Cloud oriented subgraph identification in massive social networks. In: ASONAM, pp. 248–255 (2010)
14. Sun, Z., Wang, H., Wang, H., Shao, B., Li, J.: Efficient subgraph matching on billion node graphs. PVLDB **5**(9), 788–799 (2012)
15. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007)

16. Merrill, D., Garland, M., Grimshaw, A.: Scalable GPU graph traversal. In: PPoPP, pp. 117–128 (2012)
17. Katz, G.J., Kider Jr., J.T.: All-pairs shortest-paths for large graphs on the GPU. In: GH, pp. 47–55 (2008)
18. Vineet, V., Harish, P., Patidar, S., Narayanan, P.J.: Fast minimum spanning tree for large graphs on the gpu. In: HPG, pp. 167–171 (2009)
19. Jenkins, J., Arkatkar, I., Owens, J.D., Choudhary, A., Samatova, N.F.: Lessons learned from exploring the backtracking paradigm on the GPU. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 425–437. Springer, Heidelberg (2011)
20. McGregor, J.J.: Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. Information Sciences **19**(3), 229–250 (1979)
21. Lee, J., Han, W.S., Kasperovics, R., Lee, J.H.: An in-depth comparison of subgraph isomorphism algorithms in graph databases. PVLDB **6**(2), 133–144 (2012)
22. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a MapReduce framework on graphics processors. In: PACT, pp. 260–269 (2008)
23. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating CUDA graph algorithms at maximum warp. In: PPoPP, pp. 267–276 (2011)
24. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A Recursive Model for Graph Mining. In: SDM, pp. 442–446 (2004)
25. Zhong, J., He, B.: Medusa: Simplified graph processing on GPUs. TPDS **25**(6), 1543–1552 (2013)
26. Harris, M., Sengupta, S., Owens, J.D.: Gpu gems 3. Parallel Prefix Sum (Scan) with CUDA, pp. 851–876 (2007)
27. Lu, M., He, B., Luo, Q.: Supporting extended precision on graphics processors. In: DaMoN, pp. 19–26 (2010)