Parallel Computing 78 (2018) 101–114

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Orchestrating parallel detection of strongly connected components on GPUs☆

Xuhao Chen*, Cheng Chen, Jie Shen, Jianbin Fang, Tao Tang, Canqun Yang, Zhiying Wang

College of Computer, National University of Defense Technology, China

ARTICLE INFO

Article history:
Received 1 April 2017
Revised 1 August 2017
Accepted 2 November 2017
Available online 10 November 2017

Keywords:
Strongly connected components
GPU
Real-world graphs
Hybrid parallelism

ABSTRACT

Detecting strongly connected components (SCC) is a practical graph analytics algorithm widely used in many application domains. To accelerate SCC detection, parallel algorithms have been proposed and implemented on GPUs. However, existing GPU implementations show unstable performance for various graphs, especially for real-world graphs, as these implementations do not have a clear understanding of the graph properties. In this paper, we analyze that graphs in SCC detection usually exhibit (1) skewed component sizes (the *static* property) and (2) dynamically changed graph structure (the *dynamic* property). To deal with these irregular graph properties, we propose a hybrid method that divides the algorithm into two phases and exploits different levels of parallelism for different-sized components. We also customize the graph traversal strategies for each phase to handle the dynamically changed graph structure. Our method is carefully implemented to take advantage of the GPU hardware. Evaluation with diverse synthetic and real-world graphs shows that our method substantially improves existing GPU implementations, both performance-wise and applicability-wise. It also achieves an average speedup of 5.6× and 1.5× over the sequential and OpenMP implementations on the CPU respectively.

1. Introduction

Strongly connected component (SCC) detection is a fundamental graph analysis problem that is pervasively present in many application domains. Tarjan's algorithm [1] is an efficient sequential method to solve the SCC detection. However, parallelizing Tarjan's algorithm is challenging as it applies an inherently sequential DFS (depth-first search) traversal of the graph. To accelerate the SCC detection for large-scale graphs, parallel algorithms using the BFS (breadth-first search) traversal have been proposed. The Forward-Backward (FB) algorithm [2] and its enhancement FB-Trim [3] are practical algorithms that bring in performance improvement.

Barnat *et al.* [4] implemented the FB-Trim algorithm using the CUDA programming model on the GPU. Due to the parallelization, their implementation achieves high performance for some randomly generated graphs. However, their implementation does not fully take graph properties into consideration, and therefore the implementation cannot work well for different types of graphs, especially for the real-world graphs [5,6].

☆ The source code of this work can be found at https://github.com/chenxuhao/gardenia

* Corresponding author.
E-mail addresses: chenxuhao@nudt.edu.cn, cxh@illinois.edu, chen_xuhao@126.com (X. Chen), chencheng@nudt.edu.cn (C. Chen), j.shen@nudt.edu.cn (J. Shen), canqun@nudt.edu.cn (C. Yang).

https://doi.org/10.1016/j.parco.2017.11.001
0167-8191/

On the other hand, Hong *et al.* [7] improved the FB-Trim algorithm with an efficient parallel CPU SCC detection method specifically for processing real-world graphs. They used a two-phase method to handle small-world graphs, and got tremendous speedup on multicore CPUs. Hong's work implies that graph algorithms should be aware of graph properties and make adjustment to handle different situations. Graph properties are also critical for GPU implementations as we evaluated.

Real-world graphs in social networks usually exhibit the *small-world* property with a *power-law* degree distribution, and therefore graphs usually include a giant SCC and a lot of small-sized nontrivial SCCs (i.e. skewed component sizes, the *static* graph property). In addition, the graph structure is *dynamically changed* when performing the SCC detection. That is to say, once an SCC is detected, it is removed from the original graphs (i.e. the *dynamic* graph property due to the graph algorithm). Therefore, after the giant SCC is detected and removed, the remaining graph contains a large amount of disconnected small subgraphs. Previous GPU implementations (e.g. Barnat's implementation) cannot efficiently handle such cases as they becomes almost serialized when processing the remaining subgraphs.

In this work, we propose an efficient, hybrid SCC detection method on the GPU to overcome the limitation of existing GPU implementations. Our method is designed by taking graph properties into account. First, to deal with the static property, we decompose the SCC detection into two phases: processing the giant SCC and processing the remaining small-sized nontrivial SCCs. The two phases utilizes different parallelism approaches. The single giant SCC is full of data parallelism while the large amount of small-sized SCCs can benefit from task parallelism. To enable efficient task parallelism in the second phase, we examine optimizations that previously utilized in CPU SCC and port them to the GPU. Second, to deal with the dynamic property, we further devise different BFS traversal strategies and choose the suitable one for each phase. By using the two-phase hybrid method and by customizing the graph traversal strategies, our method is able to achieve high performance for a large variety of synthetic and real-world graphs.

We validate the effectiveness and efficiency of our hybrid method using CUDA on the NVIDIA GPU. Evaluation with diverse synthetic as well as real-world graphs shows that our method significantly outperforms existing GPU implementations. We also compare our method with the state-of-the-art sequential and OpenMP implementations on the CPU, and we achieve an average speedup of $5.6\times$ and $1.5\times$, respectively.

The main contributions in this work are:

(1) We propose a hybrid SCC detection method that decomposes the SCC detection into two phases and enables different parallelism approaches for different phases to deal with graph irregularities.

(2) We examine the state-of-the-art graph traversal strategies and apply the best-performing strategy to fit the graph properties of each SCC phase.

(3) We port optimization techniques proposed in CPU SCC detection to our GPU implementation to exploit more parallelism.

(4) We demonstrate the effectiveness and efficiency of our hybrid method by implementing and evaluating the proposed method with different types of synthetic and real-world graphs.

The rest of the paper is organized as follows: Section 2 introduces the existing parallel algorithms as well as the state-of-the-art GPU implementations. Section 3 details our proposed design. The experimental evaluation is present in Section 4. We discuss related work in Section 5, and we conclude the paper in Section 6.

2. Background and motivation

A strongly connected component in a directed graph refers to a maximal subgraph where there exists a path between any two vertices in the subgraph. SCC detection which decomposes a given directed graph into a set of disjoint SCCs is widely used in many graph analytics applications, including web and social network analysis [8], formal verification [9], reinforcement learning [10], mesh refinement [3], computer-aided design [11] and scientific computing [12].

The classic sequential SCC detection algorithm, a.k.a Tarjan's algorithm, is difficult to parallelize because it is based on the DFS graph traversal which is known to be inherently sequential [13]. Thus, parallel algorithms have been investigated and designed to speedup SCC detection on parallel machines. In this section, we first introduce widely used parallel algorithms, and then discuss existing GPU implementations and their performance limitation.

2.1. Parallel SCC detection

Fleischer *et al.* proposed a practical algorithm, i.e. Forward-Backward (FB) algorithm [2], which achieves parallelism by recursively partitioning the given graph into three disjoint subgraphs that can be processed independently. McLendon *et al.* [3] extends the FB algorithm with a Trim step to quickly detect size-1[1] SCCs. The FB-Trim algorithm is shown in Algorithm 1. This algorithm includes two parts: FB and Trim.

The FB part proceeds as follows. A vertex called **pivot** p is selected (line 4) and the strongly connected component S that this pivot belongs to is computed (line 7) as the intersection of the forward reachable set FW (line 5) and backward reachable sets BW (line 6) of the pivot. Computation of the reachable sets divides the graph into four subgraphs: (1) the strongly connected component S with the pivot, (2) the subgraph $FW \setminus S$ given by vertices in the forward reachable set but

[1] Size-n SCC means the SCC contains n vertices. Thus, size-1 SCC only contains one vertex.

Algorithm 1 FB-Trim Algorithm [3].

```
1: procedure FB-Trim(G(V, E), SCC)
2:     Trim(G, SCC)
3:     if V ≠ ∅ then
4:         p ← pick any vertex in G
5:         FW ← Fwd-Reach(G, p)
6:         BW ← Bwd-Reach(G, p)
7:         S ← FW ∩ BW
8:         SCC ← SCC ∪ S
9:         in parallel do
10:            FB-Trim(FW \ S, SCC)
11:            FB-Trim(BW \ S, SCC)
12:            FB-Trim(G \ (FW ∪ BW), SCC)
13:        end in parallel
14:    end if
15: end procedure
```

not in the backward reachable set (line 10), (3) the subgraph $BW \backslash S$ given by vertices in the backward reachable set but not in the forward reachable set (line 11), and (4) the subgraph $G \backslash (FW \cup BW)$ given by vertices that are neither in the forward nor in the backward reachable set (line 12). Since an SCC cannot belong to more than one partition, each partition can be processed independently. The subgraphs that do not contain the pivot form three independent instances of the same problem, and therefore they are recursively processed in parallel with the same algorithm. Furthermore, since each subgraph produces three additional subgraphs, it is expected that quickly there would be sufficient independent tasks to consume all of the parallel processing elements [7].

Based on the FB algorithm, the FB-Trim algorithm adds a Trim step (line 2) to preprocess the *trivial SCCs* (i.e. size-1 SCCs) before picking the pivot. Since a trivial SCC has either zero incoming edges or zero outgoing edges, it can be easily identified by only looking at the number of neighbors, rather than by computing two reachable sets (which is computationally more expensive). The Trim step is described in Algorithm 2.

Algorithm 2 Trim Procedure.

```
1: procedure Trim(G(V, E), SCC)
2:     repeat
3:         for each vertex v ∈ V in parallel do
4:             if degree_in(v) = 0 or degree_out(v) = 0 then
5:                 SCC ← SCC ∪ {{v}}
6:                 G ← G \ {v}
7:             end if
8:         end for
9:     until G not changed
10: end procedure
```

2.2. Existing GPU implementations and the limitations

Barnat *et al.* [4] implemented a similar FB-Trim algorithm using CUDA [14] on the GPU[2]. Stuhl [15] improved this work with advanced graph traversal implementations. Their methods work efficiently for some randomly generated graphs, but show very limited performance when applied to graphs with many small-sized nontrivial SCCs. Fig. 1 shows the performance of Barnat's CUDA implementation normalized to the sequential Tarjan's algorithm. For non-small-world graphs which have only one nontrivial SCC (`rmat-er`, `kron21`, `Hamrle3`, `cage14` and `cage15`), Barnat's implementation achieves significant speedup ($9.5\times \sim 19.8\times$). For the remaining small-world graphs which have a giant SCC and a large amount (up to tens of thousands) of nontrivial SCCs, Barnat's implementation is much slower than the sequential algorithm.

The poor performance is due to the fact that real-world graphs usually exhibit the small-world and power-law properties, leading to skewed component sizes in SCC detection (see Fig. 2). Typically, a small-world graph, especially in social networks, contains a single giant SCC and a large amount of small-sized nontrivial SCCs. When the giant SCC is detected and removed, the remaining graph consists of a large number of disconnected subgraphs. In this case, the conventional FB-Trim algorithm becomes almost sequential because most of the subgraphs are disconnected and only a few pivots can be

[2] In Barnat's implementation, the Trim step is only applied within the main loop.

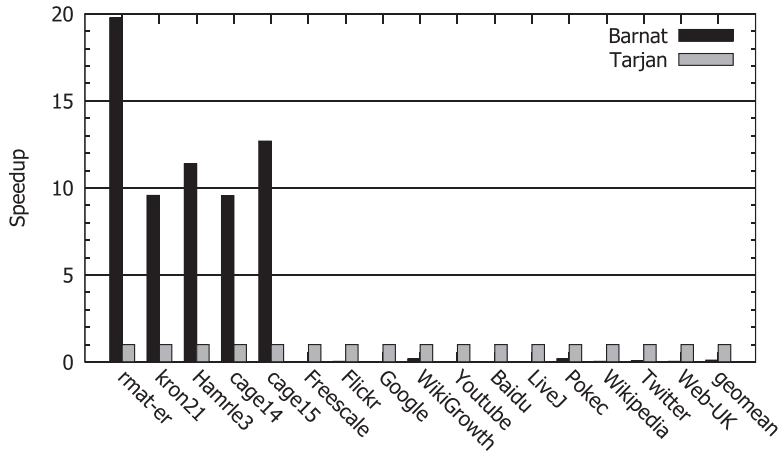

Fig. 1. Performance of Barnat's CUDA SCC detection, normalized to the sequential Tarjan's algorithm. See the datasets description in Table 1.

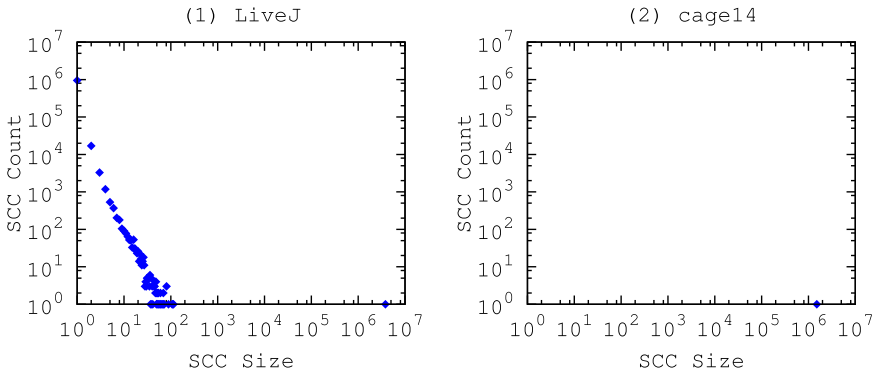

Fig. 2. The SCC size distribution of two different types of graphs. `LiveJ` (left) is a small-world graph that contains a giant SCC and lots of small-sized SCCs, while `cage14` (right) is a non-small-world graph that contains only one giant SCC (thus there is only one point in the right subfigure).

selected in each iteration. This graph irregularity is not properly considered and handled by existing GPU implementations, resulting in extreme inefficiency. Note that the problem also exists in CPU parallel implementations, but it leads to even worse performance in GPU environment, due to the weaker single-thread computation capability of the GPU.

Moreover, processing a graph with skewed component sizes requires different parallelism approaches and traversal strategies to deal with different-sized subgraphs. For example, when detecting the single giant SCC, the entire GPU is dedicated to compute it, exploiting data-level parallelism. In this phase, we can apply sophisticated graph traversal strategies. However, when processing the remaining graph with many small-sized subgraphs, straightforward strategies would be better since data parallelism is very limited and task parallelism dominates the phase. Existing GPU implementations with a fixed parallelism approach and traversal strategy can not adapt to the graph property changes, leading to severe performance degradation.

In summary, the unsatisfactory performance of existing GPU implementations motivates us to design a new method that can better handle the graph irregularities and to fully take advantage of the underlying GPU hardware.

3. Design and implementation

Despite the irregularity, recent studies [16–20] demonstrate that GPUs can substantially accelerate graph algorithms with careful design and optimization. In this section, we present our design and implementation of SCC detection that can make good use of the GPU hardware.

3.1. The design overview

Our design follows a top-down approach. First, As most real-world graphs exhibits skewed-component sizes in SCC detection (i.e., a single giant SCC and many small-sized SCCs), leading to severe irregularity and load imbalance, we propose a *hybrid* method to solve the challenge. This hybrid method divides the SCC detection into two phases, and dynamically changes parallelism approaches according to the changed graph properties (Section 3.3). After dividing the detection into two phases, we further apply the `FWCC` and `Trim2` optimizations to handle the serialization problem of the second phase,

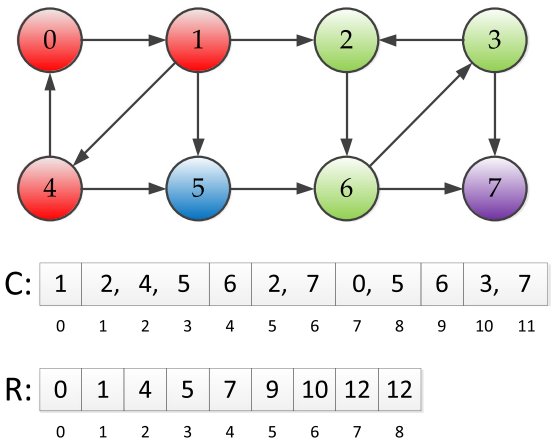

Fig. 3. An example of the compressed sparse row (CSR) format. The graph has 4 SCCs (red, green, blue, purple). The blue and purple SCCs are trvial SCCs with only one vertex. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

exploiting its parallelism (Section 3.4). In addition, as the BFS graph traversal is the most time-consuming part in each phase, we need to customize the traversal strategies for graphs of different phases. We propose and examine four traversal strategies and find the best-performing one for each phase (Section 3.5).The following sections elaborate our design and implememntation by starting with a baseline implementation (Section 3.2) and dummyTXdummy-(ending with technical details (Section 3.6).

3.2. Baseline implementation

Algorithm 3 illustrates the baseline GPU implementation. At line 2, the Trim procedure (see details in Algorithm 2) is launched to remove trivial SCCs, thus reducing the workload for the following steps. Pivot-Gen (lines 3&9) is responsible for generating pivots. A flag for each vertex is used to indicate whether the vertex is picked as a pivot. Then the main loop is launched. Fwd-Reach (line 5) and Bwd-reach (line 6) are procedures to calculate the forward and backward reachable sets of the pivots respectively. Update (line 8) is responsible for calculating the SCC (i.e. the intersection of the forward and backward reachable sets) and updating vertex status. Details of Fwd-Reach and Bwd-reach are explained in Section 3.5.

An auxiliary data structure `color` is used to help partitioning. when we partition the graph, we assign a unique `color` value to all vertices in the same subgraph; different subgraphs have different `color` values. Thus, two vertices with different `color` values are considered disconnected, even though they have an edge between them in the original graph. Besides, we use several flags to record the status of each vertex. The `removed` flag indicates the liveness of a vertex. When an SCC is identified, instead of detaching each vertex in this SCC from the rest of the graph, we simply set the `removed` flag of each vertex, and these vertices are no longer active since then. The `visited` flag is used to indicate whether the corresponding vertex has been visited during the forward and backward traversal. If a vertex is both set as forward and backward visited, it is identified as an element of the SCC that the pivot belongs to. The `expanded` flag is used to indicate whether the corresponding vertex has been expanded or not during the traversal, so that we can filter expanded vertices and remove unnecessary work.

We use the well-known compressed sparse row (CSR) sparse matrix format to store the graph in memory. Fig. 3 provides a simple example. The column-indices array C is formed from the set of the adjacency lists concatenated into a single array

Algorithm 3 Baseline GPU FB-Trim Algorithm.

```
1:  procedure FB-Trim(G(V, E), SCC)
2:      Trim(G, SCC, removed)
3:      Pivot-Gen(G, SCC, color, removed)
4:      repeat
5:          Fwd-Reach(G, SCC, color, removed, visited, expanded)
6:          Bwd-Reach(G, SCC, color, removed, visited, expanded)
7:          Trim(G, SCC, removed)
8:          Update(G, SCC, color, removed, visited, expanded)
9:          Pivot-Gen(G, SCC, color, removed, visited)
10:     until no pivot generated
11: end procedure
```

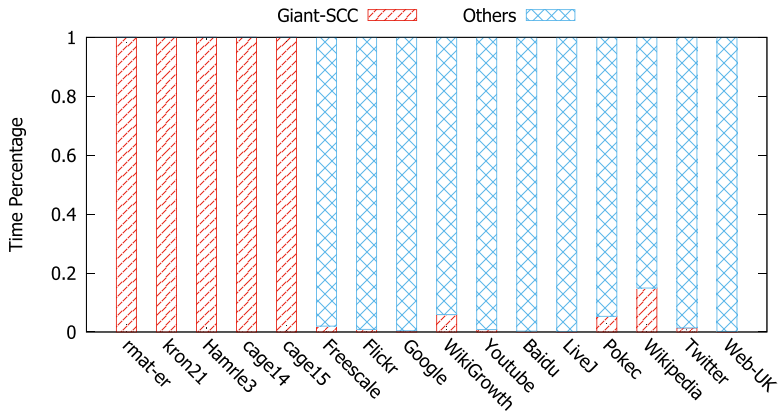

Fig. 4. Execution time distribution of Barnat's CUDA implementation.

of m integers (m is the number of edges). The row-offsets array R contains $n+1$ integers (n is the number of vertices), and entry $R[i]$ is the index in C of the adjacency list of the vertex v_i.

3.3. The hybrid method

Previous studies [4,7,21] have shown evidence that the major graph property that mostly affects the performance of SCC detection: the existence of a single giant SCC and a large amount of small-sized SCCs. This irregular property causes load imbalance and serializes the algorithm when processing small-sized SCCs, making the existing algorithm extremely inefficient.

To deal with the irregular graph property, we propose a hybrid method, in which the SCC detection problem is decomposed and solved in two phases with different parallelism approaches. In the first phase (*Phase-1*), the algorithm processes the single giant SCC with all threads, exploiting data-level parallelism. In the second phase (*Phase-2*), the remaining small-sized subgraphs are processed in parallel, exploiting task-level parallelism. We exploit different levels of parallelism to maximize the performance for each phase.

Fig. 4 illustrates the execution time distribution of Barnat's CUDA SCC detection. For non-small-world graphs, there is only one nontrivial SCC (i.e. the single giant SCC), so no Phase-2 is needed for these graphs. By contrast, for the small-world graphs, most of the time is spent on Phase-2 to process the large amount of small-sized SCCs (more than 1000) as Phase-2 is scarcely parallelized. This serialization is due to the fact that the large amount of remaining small-sized subgraphs are disconnected to each other. Recursively applying the FB-Trim algorithm to each subgraph only identifies one SCC that the pivot belongs to, and does not provide further partitioning [7]. Consequently, processing the disconnected subgraphs is almost serialized. With the serialization problem, the FB-Trim algorithm needs thousands of iterations to finish for most graphs that exhibit the small-world property. Note that although BFS within each subgraph is still parallelized, it can offer very limited parallelism since these subgraphs are small.

3.4. Exploiting parallelism in phase-2

To handle the serialization problem in Phase-2, we present FB-Trim-Hybrid that futher optimizes our GPU implementation with the adoption of the *WCC method* proposed in Hong's parallel CPU implementation. The extensions that FB-Trim-Hybrid applies to FB-Trim are: (1) finding weakly connected components (FWcc), and (2) detecting size-2 SCCs (Trim2). Both extensions are carefully mapped onto the GPU, and we will show that their overhead is well-controlled.

As mentioned in Section 3.3, after the giant SCC is identified and removed in Phase-1, Phase-2 is mostly serialized because of the disconnected small-sized SCCs. To exploit more parallelism in Phase-2, FWcc is utilized to identify weakly connected components (WCCs) before Phase-2 begins. Since one pivot is selected for each WCC, we have many pivots selected at once and we substantially improve the degree of task-level parallelism. Additionally, to reduce the execution time of FWcc, we add a Trim2 step to identify and remove size-2 SCCs before FWcc. The algorithms of FWcc and Trim2 are straightforward and thus not shown in this paper.

Algorithm 4 shows the structure of FB-Trim-Hybrid. In Phase-1 (lines $3 \sim 11$), the single giant SCC is decomposed. The transition between Phase-1 and Phase-2 occurs when an SCC containing more than 1% of the vertices of the original graph is identified (this condition often leads to the giant SCC since the rest SCCs are small ones). Next Trim2 is done to remove size-2 SCCs (line 13). Then WCCs are identified (line 14) to exploit parallelism. In Phase-2 (lines $17 \sim 23$), a large amount of small-sized SCCs are detected. Our experiments in Section 4 show that FWcc and Trim2 optimizations dramatically increase the parallelism in Phase-2, leading to significant execution time reduction of Phase-2 (see Fig. 8). Next, we try to optimize the major operations in each phase.

Algorithm 4 FB-Trim-Hybrid Algorithm.

```
1: procedure FB-Trim-Hybrid(G(V, E), SCC)
2:     /* Phase 1*/
3:     Trim(G, SCC, removed)
4:     Pivot-Gen(G, SCC, color, removed)
5:     repeat
6:         Fwd-Reach(G, SCC, color, removed, visited, expanded)
7:         Bwd-Reach(G, SCC, color, removed, visited, expanded)
8:         Trim(G, SCC, removed)
9:         Update(G, SCC, color, removed, visited, expanded)
10:        Pivot-Gen(G, SCC, color, removed, visited)
11:    until more than 1% vertices removed
12:    Trim(G, SCC, removed)
13:    Trim2(G, SCC, removed)
14:    FWcc(G, color, removed, visited)
15:    Pivot-Gen(G, SCC, color, removed, visited)
16:    /* Phase 2 */
17:    repeat
18:        Fwd-Reach(G, SCC, color, removed, visited, expanded)
19:        Bwd-Reach(G, SCC, color, removed, visited, expanded)
20:        Trim(G, SCC, removed)
21:        Update(G, SCC, color, removed, visited, expanded)
22:        Pivot-Gen(G, SCC, color, removed, visited)
23:    until no pivot generated
24: end procedure
```

3.5. Customizing graph traversal

As listed in Algorithm 3, the major SCC detection workload is the graph traversal (in Fwd-Reach and Bwd-Reach) which is implemented as parallel BFS on GPUs. Therefore it is essential to pick an efficient BFS implementation for high performance SCC detection. Parallel BFS is a well-explored field [22,23]. Basically two parallelism strategies are utilized on GPUs: *topology-driven* and *data-driven* implementations [24].

For graph algorithms, the naive topology-driven implementation simply maps each vertex to a thread, and in each iteration, the thread stays idle or is responsible to process the vertex depending on whether the corresponding vertex has been processed or not. It is straightforward to map the topology-driven implementation onto the GPU with no extra data structure. Harish *et al.* [25] first developed topology-driven BFS on GPUs. Hong *et al.* [26] improved it by mapping warps rather than threads to vertices.

By contrast, the data-driven implementation maintains a worklist which holds the vertices to be processed. In each iteration, threads are created in proportion to the worklist size (i.e. the number of vertices in the worklist). Each thread is responsible for processing a certain amount of vertices in the worklist, and no thread is idle. Therefore, the data-driven implementation is generally more work-efficient than the topology-driven one, but it needs extra overhead to maintain the worklist. Note that the data-driven implementation still suffers from load imbalance, since vertices may have different amount of edges to be processed by the corresponding threads. Merrill *et al.* [16] proposed a hierarchical load balancing strategy to deal with the problem.

We implement four versions of BFS in our SCC detection: naive topology-driven (`topo`), topology-driven with load balancing (`topo-lb`), naive data-driven (`data`), and data-driven with load balancing (`data-lb`). For `topo-lb` and `data-lb`, we use the same load balancing strategy proposed by Merrill *et al.*. Algorithm 5 illustrates the naive topology-driven im-

Algorithm 5 Forward-Reach Procedure (topology-driven).

```
1: procedure Fwd-Reach(G(V, E), color, removed, visited, expanded)
2:     repeat
3:         changed ← false
4:         for each vertex v ∈ V in parallel do
5:             Fwd-Step(G, v, color, removed, visited, expanded, changed)
6:         end for
7:     until changed = false
8: end procedure
```

Algorithm 6 Forward-Step Kernel (topology-driven).

```
1: procedure FWD-STEP(G, v, color, removed, visited, expanded, changed)
2:     if !removed(v) and visitied(v).fw and !expanded(v).fw then
3:         expanded(v).fw ← true
4:         for each vertex w ∈ adj(v) do
5:             if !removed(w) and !visited(w).fw and color(w) = color(v) then
6:                 visited(w).fw ← true
7:                 changed ← true
8:             end if
9:         end for
10:    end if
11: end procedure
```

plementation of the FWD-REACH procedure. A flag *changed* is used to indicate whether all the vertices are colored or not. This flag is cleared at the beginning of each iteration, and set by one or more threads if any vertex is updated. Once all the vertices have been visited, the flag remains *false* and the algorithm finally terminates. Algorithm 6 illustrates the FWD-STEP kernel operations.

Algorithm 7 shows the naive data-driven BWD-REACH procedure. It is implemented through worklists. At the beginning (line 2), generated pivots are pushed into the shared worklist W_{in}. Every worker thread in the system grabs a vertex from the worklist and starts performing BFS concurrently with respect to other worker threads. The program is finished when all the worker threads become idle and no work items remain in the worklist. *Double buffering* [24] is used to avoid copying the worklist. Algorithm 8 illustrates the BWD-STEP kernel operations. Note that the data structure `expanded` is not useful for the data-driven implementation.

Fig. 5 and Fig. 6 compare the performance of using these four BFS implementations in Phase-1 and Phase-2, respectively. In Fig. 5, we observe that without load balancing, `topo` outperforms `data` in most cases, since `data` has extra overhead caused by maintaining the worklist. After applying load balancing, both versions get performance improvement. For most graphs, `topo-lb` shows significant speedups (up to 3.4×). For `Baidu` and `Wikipedia`, `data-lb` performs better than `topo-lb` mainly due to its work-efficiency. On average, `topo-lb` achieves the best performance among the four, with a geomean speedup of 1.5× over `topo`.

Fig. 5 demonstrates that load balancing can accelerate BFS when processing the largest SCC in Phase-1. However, for Phase-2 where many small disconnected subgraphs exists, Fig. 6 illustrates that load balancing is not effective since its overhead exceeds its performance benefits, although some graphs (e.g. `Wikigrowth` and `Wikipedia`) can still benefit from load balancing and get speedup with `data-lb`.

In summary, according to the observation, we decide to apply `topo-lb` in Phase-1 and switch to `topo` in Phase-2.

Algorithm 7 Backward-Reach Procedure (data-driven).

```
1: procedure BWD-REACH(G^T(V, E), color, removed, visited, expanded)
2:     W_in ← pivots
3:     while W_in ≠ ∅ do
4:         for each vertex v ∈ W_in in parallel do
5:             BWD-STEP(G, v, color, removed, visited, W_out)
6:         end for
7:         swap(W_in, W_out)                          ▷ Swap the worklists
8:         W_out ← ∅
9:     end while
10: end procedure
```

Algorithm 8 Backward-Step Kernel (data-driven).

```
1: procedure BWD-STEP(G, v, color, removed, visited, W_out)
2:     for each vertex w ∈ adj(v) do
3:         if !removed(w) and !visited(w).bw and color(w) = color(v) then
4:             visited(w).bw ← true
5:             W_out ← W_out ∪ {w}                    ▷ Atomic push
6:         end if
7:     end for
8: end procedure
```

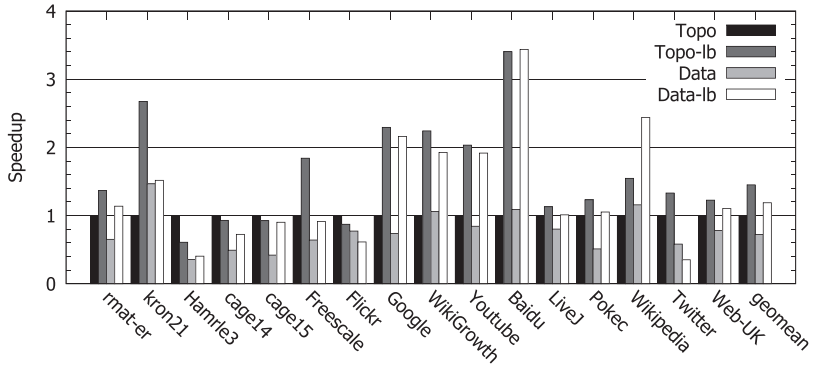

Fig. 5. Performance of topology-driven v.s. data-driven implementations when processing the single giant SCC in Phase-1, all normalized to the `topo` implementation.

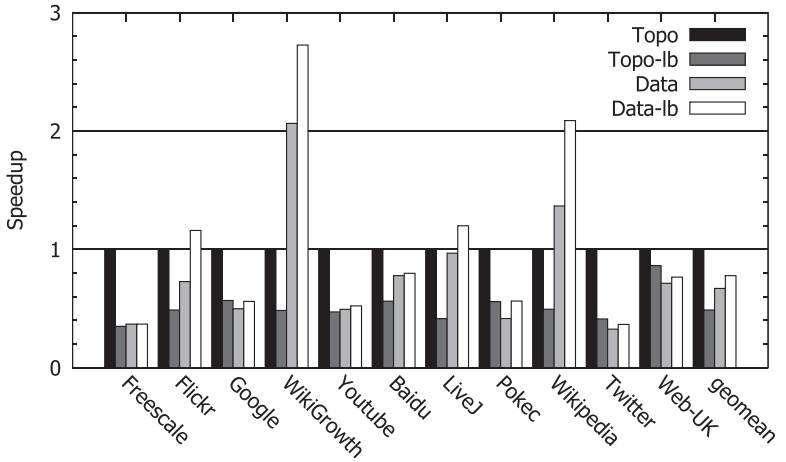

Fig. 6. Performance of topology-driven v.s. data-driven implementations when processing the small-sized SCCs in Phase-2, all normalized to the `topo` implementation. Note that we only show graphs that have many small-sized SCCs to be processed in Phase-2.

3.6. Technical details

Status Update. In real implementation, we use a single data structure `status` to hold all the flags including `removed`, `visited`, `expanded`, `is_pivot` and so on. Each bit in `status` represents a distinct flag. Thus, we use bitset operations instead of boolean operations to update status. This implementation reduces storage space as well as the number of reads and writes on these flags. Note that our UPDATE routine is also responsible for color assignment according to the results of forward and backward traversal.

Pivot Generation. Typically, pivots are generated by a pseudo random number generator. However, since multiple subgraphs are processed simultaneously in the same CUDA kernel, we need to choose a number of pivots, one for each subgraph. Barnat *et al.* proposed to let all vertices of a subgraph concurrently write their own unique identifiers to a single memory location [4]. The vertex that wins the competition will be selected as the pivot of its subgraph. Our PIVOT-GEN routine uses this method to generate pivots, and then sets the status (e.g. `visited`) for the selected pivots.

4. Evaluation

In this section, we evalute our proposed method with various graph datasets (listed in Table 1). We use the R-MAT [27] graph generator GTGraph [28] to generate `rmat-er` by using the parameters (0:25; 0:25; 0:25; 0:25). We choose `kron21` from the 10th DIMACS Implementation Challenge (generated by the kronecker generator). We also pick real-world graphs from the University of Florida Sparse Matrix Collection [29], the SNAP database [30], and the Koblenz Network Collection [31]. These graphs are also used in previous work [7,21,32]. In summary, we use 2 synthetic graphs and 14 real-world graphs for our evaluation. The graphs vary widely in size, degree distribution, density of local subgraphs and application domain.

Table 1
Suite of benchmark graphs. * indicates that the original graph is undirected; we randomly assign a direction for each edge with 50% probability for each direction.

Graph	# Vertices	# Edges	Largest SCC Size	Avg. deg.	Description
rmat-er	4,194,304	4,194,304	4,194,304	10.0	Synthetic random graph
kron21*	2,097,152	91,042,010	1,180,037	43.4	Synthetic random graph
Hamrle3	1,447,360	5,514,242	1,447,360	3.8	Circuit simulation
cage14	1,505,785	27,130,349	1,505,785	18.0	DNA electrophoresis
cage15	5,154,859	99,199,551	5,154,859	19.2	DNA electrophoresis
Freescale	2,999,349	23,042,677	2,888,522	7.7	Circuit simulation
Flickr	2,302,925	33,140,017	1,605,184	14.4	Connection of Flickr users
Google	875,713	5,105,039	434,818	5.8	Web graph from Google
WikiGrowth	1,870,709	39,953,145	1,629,321	21.4	English Wikipedia with edge arrival times
Youtube	1,138,499	4,942,297	509,245	4.3	Youtube users and their connections
Baidu	2,141,300	17,794,839	609,905	8.3	Chinese online encyclopedia Baidu
LiveJ	4,847,571	68,993,773	3,828,682	14.2	LiveJournal online social network
Pokec	1,632,803	30,622,564	1,304,537	18.8	Pokec online social network
Wikipedia	3,148,440	39,383,235	2,104,115	12.5	Links in Wikipedia pages
Twitter*	21,297,772	265,025,809	10,351,983	12.4	Connection of Twitter users
Web-UK*	18,520,343	298,113,762	14,479,249	16.1	Web crawl of .uk domain

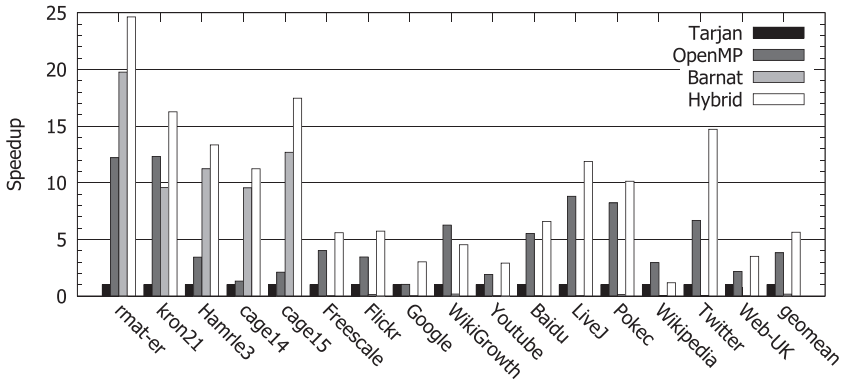

Fig. 7. Performance of the SCC detection implementations, all normalized to the sequential Tarjan's algorithm.

4.1. Experiment setup

We compare 4 implementations including (1) `Tarjan`: Tarjan's serial algorithm implemented in [4], (2) `OpenMP`: Hong's OpenMP implementation [7], (3) `Barnat`: Barnat's CUDA implementation [4], (4) `Hybrid`: our proposed GPU implementation FB-Trim-Hybrid. We conduct the experiments on the NVIDIA K80 GPU with CUDA Toolkit 8.0 release. `Tarjan` and `OpenMP` is executed on the Intel Xeon E5 26790V2 2.30 GHz CPU with 12 cores. We launch 12 threads for `OpenMP` since this is the best performing configuration as we have evaluated. We use `gcc` and `nvcc` with the `-O3` optimization option for compilation along with `-arch=sm_35` when compiling for the GPU. We execute all the benchmarks 10 times and collect the average execution time to avoid system noise. Timing is only performed on the computation part of each program. For all the GPU implementations, the input/output data transfer time (usually takes 10%-15% of the entire program execution time) is excluded.

4.2. Overall performance

Fig. 7 compares the performance of our proposed FB-Trim-Hybrid method with `Tarjan`, `OpenMP` and `Barnat`. On average, our implementation achieves the best performance among the four methods. `Hybrid` obtains a geomean speedup of 5.6× compared to the Tarjan's serial one, while `OpenMP` gets 3.8× performance improvement. Compared to `OpenMP`, our method is 47% faster. This speedup over `OpenMP` is reasonable because the CPU has a much larger last level cache which can better capture locality than that on the GPU, although the GPU has higher throughput and memory bandwidth.

As mentioned in Section 2.2, `Barnat` gets speedup for the first five graphs (because these graphs have only one nontrivial SCCs), but it is much slower than `Tarjan` on average. By contrast, our method consistently works better than `Tarjan` and `Barnat`. For the first five graphs, `Hybrid` is faster than `Barnat` thanks to the optimized BFS implementation (see details in Section 3.5), while for the rest small-world graphs, our method outperforms `Tarjan` and `Barnat` mainly because of the much higher parallelism exploited by the WCC method. (see Section 3.4). Table 2 shows that FWcc substantially reduces the number of iterations required to complete the SCC detection. For small-world graphs, without FWcc, `Barnat` needs

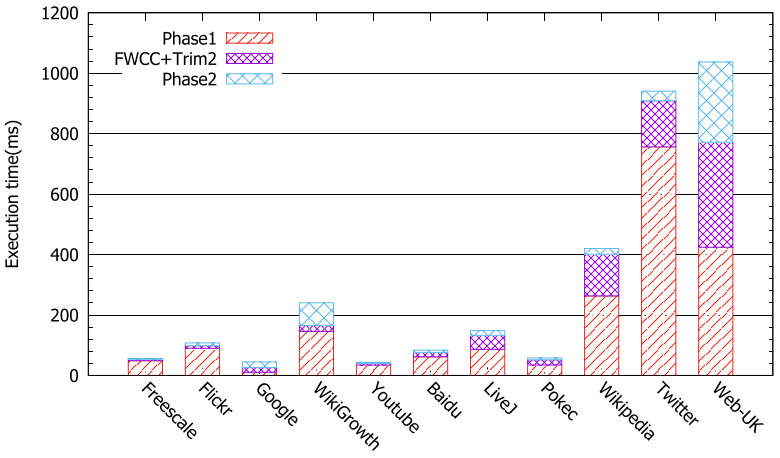

Fig. 8. Execution time breakdown of our proposed FB-Trim-Hybrid implementation. Note that we only show graphs that have many small-sized nontrivial SCCs.

Table 2
The number of iterations required to complete SCC detection for each graph. The third column lists the number of nontrivial SCCs in each graph.

Graphs	Barnat	Hybrid	# nontriv.
rmat-er	1	1	1
kron21	1	1	1
Hamrle3	1	1	1
cage14	1	1	1
cage15	1	1	1
Freescale	55,084	2	55084
Flickr	28,804	6	58636
Google	5347	14	12874
WikiGrowth	2702	4	2835
Youtube	10,752	6	11370
Baidu	9371	5	22282
LiveJ	12,226	5	23456
Pokec	1080	3	2094
Wikipedia	2559	5	2666
Twitter	18,223	5	18491
Web-UK	81,638	31	56119

thousands of iterations to finish since its Phase-2 is almost sequential. By contrast, our method terminates within several iterations. In general, our GPU method is more practical and efficient than the existing parallel implementations.

4.3. Execution time breakdown

To better understand the performance impact of our optimizations, we breakdown the execution time into three parts: Phase-1, FWcc+Trim2 and Phase-2, shown in Fig. 8. As expected, since FWcc exploits more parallelism, execution time spent in Phase-2 is significantly reduced, and Phase-2 does not dominate the total execution time any more (see Fig. 4 for comparison). We find that FWcc increases the Phase-2 performance by 9.0×, and adding Trim2 further increases the performance by 1.4×.

Meanwhile, with refined BFS traversal strategies, the Phase-1 performance is also improved. Besides, we parallelize our FWcc implementation on the GPU and ensure its low overhead. To sum up, orchestrating all the three parts with customized optimizations transforms into the final performance improvement.

4.4. Hybrid vs. OpenMP

We note that our method outperforms OpenMP for all evaluated graphs except Wikipedia and WikiGrowth, so we look into the execution time breakdown of OpenMP and Hybrid. We notice that the performance gap is mainly due to the fact that OpenMP spends less time on Phase-1 than Hybrid does (see Fig. 9(a)). By examining the BFS traversal of the two graphs, we find that both graphs have a long tail that needs processing in the last tens or hundreds of iterations (see Fig. 9(b)). In the long tail part, the frontier size (the number of active vertices) is small but the traversal needs hundreds rounds of iterations to terminate. Using the topo-lb strategy cannot efficiently process the long tail part, as it needs to

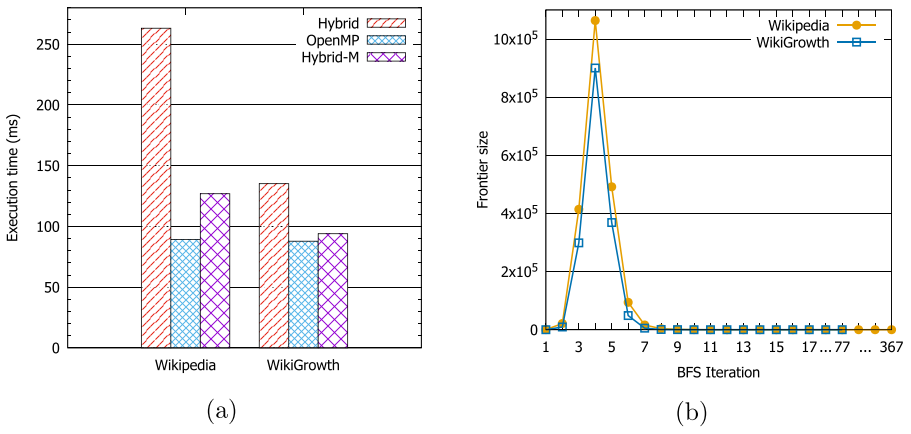

Fig. 9. (a) Execution time of Phase-1 of `OpenMP` and `Hybrid` for `Wikipedia` and `WikiGrowth`. The bars `Hybrid-M` shows the execution time after applying the `mixed` top-down and bottom-up strategy in our hybrid implementation. (b) The two graphs both have a long tail with tens or hundreds of iterations.

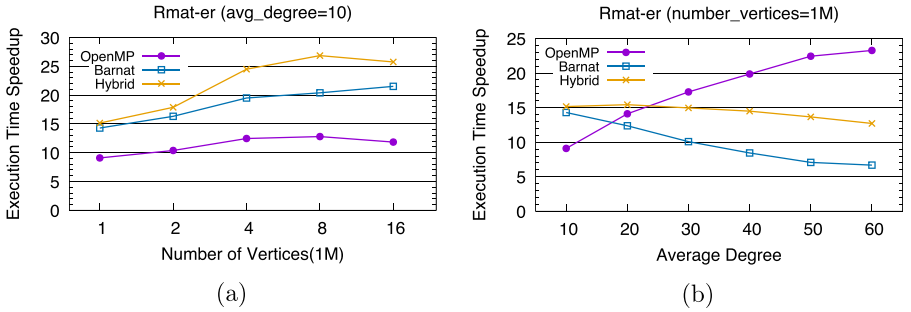

Fig. 10. (a) Performance of the SCC detection with varied dataset size (1M to 16M vertices); (b) Performance of the SCC detection with varied graph density; all normalized to the sequential Tarjan's algorithm.

scan through all the vertices in every iteration. Beamer *et al.* proposed a mixed top-down and bottom-up approach [22] to deal with the long-tail problem on the CPU. We apply a similar `mixed` traversal strategy (instead of `topo-lb`) for the two graphs on the GPU. The bars `Hybrid-M` in Fig. 9(a) shows that after applying the `mixed` strategy, the execution time of the Phase-1 is largely reduced (and close to that in `OpenMP`).

4.5. Sensitivity study

We first evaluate the sensitivity of our method when changing the size of the input datasets. In this experiment, we change the graph size from 1M to 16M vertices, with a fixed density (average degree) of 10. Fig. 10(a) shows the execution time speedup over the Tarjan's sequential method. It is clear that our method consistently outperforms the `OpenMP` and `Barnat` implementations as the input scale increases. For `OpenMP` and `Hybrid`, the performance speedup increases as the size increase from 1M to 8M. This is as expected since larger datasets would benefit more from parallel implementations. The speedup drops a little bit when the size goes to 16M, possibly due to the graph topology, but `Hybrid` still achieves a significant speedup of 25.8× over `Tarjan`. By contrast, `Barnat` shows limited performance superiority compared to `OpenMP`. Note that we focus on single-GPU implementation in this paper, while our work can be extended to multi-GPU machines when the graph sizes exceed a single GPU's memory.

We also evaluate the effect of graph density on performance. Fig. 10(b) illustrates how the performance changes as the graph density increases. In this experiment, we change the graph density from 10 to 60, with a fixed graph size of 1M vertices. We observe that `Hybrid` still consistently outperforms `Barnat`. This performance gap is almost unchanged as the graph becomes denser. `OpenMP` exhibits higher performance speedup with denser input graphs, while the speedups of the two GPU implementations drop as the density increases. This is possibly due to the much larger cache size of the CPU. Since the working set size is proportional to the graph density, GPUs are likely to suffer higher degree of memory divergence and cache thrashing with limited on-chip cache size. The observation suggests us optimize cache behavior so as to further improve the performance of GPU SCC detection. Note that real-world graphs are usually sparse graphs, e.g. the largest density of the real-world graph instances used in our experiments is 21, and our GPU method consistantly achieves better performance than the CPU parallel implementation for real-world sparse graphs.

5. Related work

Parallel SCC detection is an important graph analysis algorithm that has been intensively studied previously. As mentioned, Hong *et al.* were the first to use the WCC method to handle small-world graphs, and Barnat *et al.* were the first to implement FB-Trim algorithm on GPUs. Inspired by Hong's work, Slota *et al.* [21] proposed a `Multistep` strategy to deal with small-world graphs on CPUs. Their approach combines BFS and coloring-based methods and uses them in different algorithm steps. Slota *et al.* [32] also implemented their `Multistep` method on GPUs to handle the large amount of small-sized SCCs in the real-world graphs. Instead of using coloring algorithm, our GPU implementation imports Hong's WCC method to handle this problem. More importantly, our method enables the adoption of different graph traversal strategies for different algorithm phases.

Many other graph algorithms have been developed on GPUs. Hong *et al.* [26] proposed a warp-centric method that maps warps rather than threads to vertices to deal with load imbalance. Merrill *et al.* [16] developed a data-driven BFS on GPUs. They employed (1) prefix sum to reduce atomic operations and (2) dynamic load balancing to deal with power-law graphs. These two techniques are both applicable to our implementation, while our work focuses more on the algorithm-specific refinement, e.g. hybrid parallelism that alleviates side effects of graph irregularity. All the work demonstrates that, with careful mapping and optimizations, graph algorithms can get substantial performance boost on GPUs. Our work further enhances the conclusion of previous practices, while we show the importance of both algorithm-specific and architecture-specific optimizations for graph analytics problems.

This article is an extension of our previous workshop paper [33]. We add a detailed analysis of the graph properties and refine the algorithm description. We also enhance the evaluation with a more solid experimental setup (e.g. larger datasets) and more comprehensive performance comparison, leading to a deeper and clearer understanding of the experimental results.

6. Conclusion

SCC detection is an important graph algorithm that has been applied in many application domains. Existing GPU implementations cannot efficiently process different types of graphs because the implementations are not aware of the graph properties. In this paper, we demonstrate that it is of great importance to understand the graph properties for accelerating SCC detection. There are two types of properties: (1) the static property, i.e. the small-world and power-law property which leads to skewed SCC sizes, and (2) the dynamic property, i.e. the dynamically changed graph structure due to the intrinsic nature of the SCC detection algorithm.

To deal with the static property, we propose a hybrid method that divides the algorithm into two phases (for processing the single giant SCC and many small-sized nontrivial SCCs respectively) and utilizes the most suitable parallelism approach for each phase. To deal with the dynamic property, we customize the graph traversal strategies to adjust to the runtime graph structure. Experiments with different types of synthetic and real-world graphs shows that the proposed method largely outperforms existing GPU implementations, and is applicable to different types of graphs. Our work further helps the graph analytics community to better understand graph algorithm acceleration on modern massively parallel processors.

Acknowledgment

We thank the anonymous reviewers for the insightful comments and suggestions. This work is partly supported by the National Natural Science Foundation of China (NSFC) No. 61502514, No. 61402488, and No. 61602501, and the National Key Research and Development Program of China under grant No. 2016YFB0200400.

References

[1] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.
[2] L. Fleischer, B. Hendrickson, A. Pinar, On identifying strongly connected components in parallel, in: Proceedings of the 15th IPDPS Workshops, IPDPS '00, Springer-Verlag, London, UK, UK, 2000, pp. 505–511.
[3] W. McLendon III, B. Hendrickson, S.J. Plimpton, L. Rauchwerger, Finding strongly connected components in distributed graphs, J. Parallel Distributed Comput. (JPDC) 65 (8) (2005) 901–910.
[4] J. Barnat, P. Bauch, L. Brim, M. Ceska, Computing strongly connected components in parallel on cuda, in: Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS), IPDPS '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 544–555.
[5] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, J. Wiener, Graph structure in the web, Comput. Networks 33 (1–6) (2000) 309–320.
[6] A. Mislove, M. Marcon, K.P. Gummadi, P. Druschel, B. Bhattacharjee, Measurement and analysis of online social networks, in: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, IMC '07, ACM, New York , NY, USA, 2007, pp. 29–42.
[7] S. Hong, N.C. Rodia, K. Olukotun, On fast parallel detection of strongly connected components (scc) in small-world graphs, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), SC '13, ACM, New York, NY, USA, 2013. 92:1–92:11
[8] R. Kumar, J. Novak, A. Tomkins, Structure and evolution of online social networks, in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), KDD '06, ACM, New York, NY, USA, 2006, pp. 611–617.
[9] R. Hojati, R.K. Brayton, R.P. Kurshan, Bdd-based debugging of design using language containment and fair ctl, in: Proceedings of the 5th International Conference on Computer Aided Verification, CAV '93, Springer-Verlag, London, UK, UK, 1993, pp. 41–58.
[10] S.J. Kazemitabar, H. Beigy, Automatic discovery of subgoals in reinforcement learning using strongly connected components, in: Proceedings of the 15th International Conference on Advances in Neuro-information Processing - Volume Part I, ICONIP'08, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 829–834.

[11] A. Xie, P.A. Beerel, Implicit enumeration of strongly connected components and an application to formal verification, Trans. Comp.-Aided Des. Integ. Cir. Sys. 19 (10) (2006) 1225–1230.
[12] A. Pothen, C.-J. Fan, Computing the block triangular form of a sparse matrix, ACM Trans. Math. Softw. (TOMS) 16 (4) (1990) 303–324.
[13] J.H. Reif, Depth-first search is inherently sequential, Inf. Process Lett. 20 (5) (1985) 229–234.
[14] NVIDIA, 2015, CUDA C Programming Guide v7.0.
[15] M. Stuhl, Computing Strongly Connected Components With CUDA. Master Thesis, Masaryk University, 2013.
[16] D. Merrill, M. Garland, A. Grimshaw, Scalable gpu graph traversal, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), PPoPP '12, ACM, New York, NY, USA, 2012, pp. 117–128.
[17] A. McLaughlin, D.A. Bader, Scalable and high performance betweenness centrality on the gpu, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), SC '14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 572–583.
[18] A. Davidson, S. Baxter, M. Garland, J.D. Owens, Work-efficient parallel gpu methods for single-source shortest paths, in: Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS), 2014, pp. 349–359.
[19] G.M. Slota, S. Rajamanickam, K. Madduri, Parallel graph coloring for manycore architectures, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 1–10.
[20] P. Li, X. Chen, Z. Quan, J. Fang, H. Su, T. Tang, C. Yang, High performance parallel graph coloring on gpgpus, in: Proceedings of the 30th IPDPS Workshop, IPDPSW '16, 2016, pp. 1–10.
[21] G.M. Slota, S. Rajamanickam, K. Madduri, Bfs and coloring-based parallel algorithms for strongly connected components and related problems, in: Proceedings of IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS), 2014, pp. 550–559.
[22] S. Beamer, K. Asanović, D. Patterson, Direction-optimizing breadth-first search, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012. 12:1–12:10
[23] V. Agarwal, F. Petrini, D. Pasetto, D.A. Bader, Scalable graph exploration on multicore processors, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), SC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–11.
[24] R. Nasre, M. Burtscher, K. Pingali, Data-driven versus topology-driven irregular computations on gpus, in: Proceedings of the 27th IEEE International Parallel Distributed Processing Symposium (IPDPS), IPDPS '13, 2013, pp. 463–474.
[25] P. Harish, P.J. Narayanan, Proceedings of the 14th international conference high performance computing (hiPC), in: Ch. Accelerating Large Graph Algorithms on the GPU Using CUDA, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 197–208.
[26] S. Hong, S.K. Kim, T. Oguntebi, K. Olukotun, Accelerating cuda graph algorithms at maximum warp, in: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), PPoPP '11, ACM, New York, NY, USA, 2011, pp. 267–276.
[27] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-MAT: A recursive model for graph mining, in: SDM, SIAM, 2004.
[28] K. Madduri, D.A. Bader, 2006,. GTgraph: A suite of synthetic graph generators. http://www.cse.psu.edu/madduri/software/GTgraph/.
[29] The University of Florida Sparse Matrix Collection, 2011. URL http://www.cise.ufl.edu/research/sparse/matrices/.
[30] J. Leskovec, 2013,. Snap: Stanford network analysis platform http://snap.stanford.edu/data/index.html.
[31] Koblenz network collection, 2013. URL http://konect.uni-koblenz.de.
[32] G.M. Slota, S. Rajamanickam, K. Madduri, High-performance graph analytics on manycore processors, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2015, pp. 17–27.
[33] P. Li, X. Chen, J. Shen, J. Fang, T. Tang, C. Yang, High performance detection of strongly connected components in sparse graphs on gpus, in: In the Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores, in conjunction with the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), PMAM '17, 2017, pp. 1–10.