High Performance Detection of Strongly Connected Components in Sparse Graphs on GPUs

Pingfan Li
College of Computer, National University of Defense Technology
Changsha 410073, China

Xuhao Chen*
College of Computer, National University of Defense Technology
Changsha 410073, China

Jie Shen
College of Computer, National University of Defense Technology
Changsha 410073, China

Jianbin Fang
College of Computer, National University of Defense Technology
Changsha 410073, China

Tao Tang
College of Computer, National University of Defense Technology
Changsha 410073, China

Canqun Yang
College of Computer, National University of Defense Technology
Changsha 410073, China

ABSTRACT

Detecting strongly connected components (SCC) has been broadly used in many real-world applications. To speedup SCC detection for large-scale graphs, parallel algorithms have been proposed to leverage modern GPUs. Existing GPU implementations are able to get speedup on synthetic graph instances, but show limited performance when applied to large-scale real-world datasets. In this paper, we present a parallel SCC detection implementation on GPUs that achieves high performance on both synthetic and real-world graphs. We use a hybrid method that divides the algorithm into two phases. Our method is able to dynamically change parallelism strategies to maximize performance for each algorithm phase. We then orchestrates the graph traversal kernel with customized strategy for each phase, and employ algorithm extensions to handle the serialization problem caused by irregular graph properties. Our design is carefully implemented to take advantage of the GPU hardware. Evaluation with diverse graphs on the NVIDIA K20c GPU shows that our proposed implementation achieves an average speedup of 5.0× over the serial Tarjan's algorithm. It also outperforms the existing OpenMP implementation with a speedup of 1.4×.

CCS CONCEPTS

•Computing methodologies → Massively parallel algorithms;

KEYWORDS

Strongly Connected Components; GPU; Real-world Graphs

ACM Reference format:
Pingfan Li, Xuhao Chen, Jie Shen, Jianbin Fang, Tao Tang, and Canqun Yang. 2017. High Performance Detection of Strongly Connected Components in Sparse Graphs on GPUs. In *Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores, Austin, TX, USA, February 04-08, 2017 (PMAM'17),* 10 pages.

*Corresponding author: cxh@illinois.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PMAM'17, February 04-08, 2017, Austin, TX, USA
© 2017 ACM. ISBN 978-1-4503-4883-6/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3026937.3026941

1 INTRODUCTION

Strongly connected component (SCC) detection is a fundamental graph analysis problem that is pervasively present in many application domains. Tarjan's algorithm is an efficient sequential method to solve SCC detection. However, parallelizing this algorithm is challenging since it uses the inherently sequential depth-first search (DFS) traversal of the graph. To speedup SCC detection for large-scale graphs, parallel algorithms have been proposed. The Forward-Backward (FB) algorithm [10] and its enhancement FB-Trim [22] are practical algorithms that bring in performance improvement.

Barnat *et al.* [4] implemented the FB-Trim algorithm using CUDA on the GPU. Although they achieve significant speedups for synthetic graphs, their implementation works poorly when applied to real-world graphs [6, 24]. This is because many real-world graphs exhibit the power-law property which complicates the problem. Specifically, small-world graphs in social networks usually include a single giant SCC and a lot of small-sized nontrivial SCCs. When the giant SCC is detected and removed, the remaining graph contains a large amount of disconnected subgraphs. The conventional FB-Trim algorithm becomes almost serialized when processing the remaining graph. Besides, since the giant SCC is full of data parallelism while the remaining graph mostly benefits from task parallelism, they might need different parallelism strategies to fully take advantage of the underlying GPU hardware. These problems which do not exist in randomly generated graphs are not specifically handled in previously proposed GPU implementations.

In this work, we present a high performance SCC detection method on the GPU that efficiently processes both synthetic and real-world graphs. Our implementation is based on the FB-Trim algorithm, and extends it to handle the irregularity of real-world graphs. More specifically, we propose a hybrid method to enable the adoption of different parallelism strategies to handle different graph properties that appear in different algorithm phases. The parallelism strategies of graph traversal are customized in each phase to maximize performance. We also apply the optimization techniques proposed in CPU SCC detection to deal with the serialization problem by finding weakly connected components. We implement our proposed method using CUDA on the NVIDIA GPU. Evaluation on diverse synthetic as well as real-world graphs shows that our method significantly outperforms existing GPU implementations. The main contributions are:

1) We present a hybrid method that enables adoption of different parallelism strategies for different graph properties.

2) We examine the state-of-the-art graph traversal optimizations and apply the best-performing strategy for each algorithm phase.

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

4) We demonstrate the effectiveness and efficiency of the hybrid method by implementing and evaluating the proposed algorithm and optimizations on the NVIDIA GPU.

The rest of the paper is organized as follows: the existing parallel algorithms as well as the state-of-the-art GPU implementations are introduced in Section 2. Our proposed design is described in Section 3. We present the experimental results in Section 4. Section 5 discusses related work, and Section 6 concludes.

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 alanytics applications, including web and social network analysis [16], formal verification [12], reinforcement learning [15], mesh refinement [22], computer-aided design [34] and scientific computing [27].

The classic sequential algorithm for SCC detection, a.k.a Tarjan's algorithm [33], is difficult to parallelize because it is based on DFS graph traversal which is known to be inherently sequential [28]. Thus, parallel algorithms have been investigated to speedup SCC detection on parallel machines. In this section, we first introduce the widely used parallel algorithms, and then discuss existing GPU implementations and their performance limitations when processing real-world graph instances.

2.1 Parallel SCC Detection

Fleischer *et al.* proposed a practical algorithm, i.e. Forward-Backward (FB) algorithm [10], which achieves parallelism by recursively partitioning the given graph into three disjoint subgraphs that can be processed independently. McLendon *et al.* [22] extends FB algorithm with a Trim step which detects size-1 SCCs to improve performance. The FB-Trim algorithm is shown in Algorithm 1. This algorithm includes two parts: FB and Trim.

The FB algorithm 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 not in the backward reachable set (line 10), (3) the subgraph $BW \setminus S$ given by vertices in the backward reachable set but not in the forward reachable set (line 11), and (4) the subgraph $G \setminus (FW \cup BW)$ 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,

Algorithm 1 FB-Trim Algorithm [22]

```
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
```

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
```

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 in a system [14].

Based on the FB algorithm, the FB-Trim algorithm adds a Trim step (line 2) to preprocess the *trivial SCCs* (i.e., SCC of size one) before picking the pivot. Since a trivial SCC has either zero incoming edges or zero outgoing edges, it can be easily identified only by 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.

2.2 GPU SCC Detection

Barnat *et al.* [4] implemented the FB-Trim algorithm using CUDA [26] on the GPU. Stuhl [32] improved this work with advanced graph traversal implementations. Their methods work efficiently with randomly generated graph instances, but show very limited performance when applied to real-world graphs with many small sized nontrivial SCCs. Fig. 1 shows the performance of their CUDA implementation normalized to the sequential Tarjan's algorithm. For the three synthetic graphs (`rmat-er`, `rmat-g` and `rmat-b`), it achieves significant speedup (6× ∼ 12×), but when applied to real-world graphs (the other bars on the right) the performance is

unsatisfactory. The only one real-world graph that is accelerated by this GPU implementation is `cage14` which has only one giant nontrivial SCCs. For the other ten real-world graphs, Barnat's method is much slower than the sequential Tarjan's algorithm.

The poor performance is due to the fact that many real-world graphs exhibit irregular structural properties such as skewed component sizes. Typically, a small-world graph in social networks contains a single giant SCC and many 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 only a few pivots can be selected in each iteration due to the fact that subgraphs are disconnected. This irregular characteristic is not properly handled by existing GPU implementations, resulting in extreme inefficiency. Note that this 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 GPUs.

On the other hand, processing a graph with skewed component sizes requires different parallelism 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 algorithm 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 in this phase. Therefore, existing GPU implementations with fixed parallelism strategy can not fully take advantage of the underlying GPU hardware. The unsatisfactory performance of existing GPU implementations motivates us to apply algorithm enhancements and optimization techniques to handle graph irregularity and better leverage the GPU architecture.

3 DESIGN AND IMPLEMENTATION

Graph algorithms are considered to be difficult to parallelize on GPUs due to their irregularity [7]. However, recent works [9, 18, 21, 23, 31] demonstrate that GPUs are capable to substantially accelerate graph algorithms if the algorithms are carefully designed and optimized for the GPU architecture. In this section we first present the baseline design and basic data structures. We then propose the hybrid method with a 2-phase algorithm structure that enables dynamic changing of parallelism strategies according to the graph property. Next we apply the algorithm extensions to increase the parallelism in Phase-2. We also discuss the effect of different strategies to implement the *graph traversal*, i.e. breadth first search (BFS). Finally, we discuss some implementation details that affect the performance of GPU SCC detection. We conduct the following analyses using the NVIDIA Tesla K20c GPU.

3.1 Baseline Design

Our baseline GPU implementation of the FB-Trim algorithm is similar to Barnat's implementation, but adds a Trim step before the main loop. Algorithm 3 illustrates the algorithm skeleton. 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. Note that a data structure `mark` is added and passed to *Trim*. *Pivot-Gen* (line 3&8) is responsible for generating pivots. A status bit is used to indicate whether the corresponding vertex is marked as a pivot. Then the main loop is launched. *Fw-Reach* (line 5) and *Bw-reach* (line 6) are procedures to calculate the forward and backward reachable sets of the pivots respectively. *Update* (line 9) is responsible for calculating the SCC (i.e. the intersection of the forward and backward reachable sets) and updating vertex status. Details of *Fw-Reach* and *Bw-reach* are explained in Section 3.4.

Similar to Hong's CPU implementation [14], two auxiliary data structures are used: `mark` and `color`. When the SCC of a vertex is identified, instead of detaching the vertex from the rest of the graph, we simply set the `mark` value of the vertex to *true*, and the vertex is considered detached thereafter. Similarly, when we partition the graph, we assign the same `color` value to vertices belonging to the same subgraph; each subgraph is assigned a unique `color` value. Therefore, two vertices of different `color` values are considered disconnected, even when there exists an edge between them in the original graph. Another data structure `visited` is used to

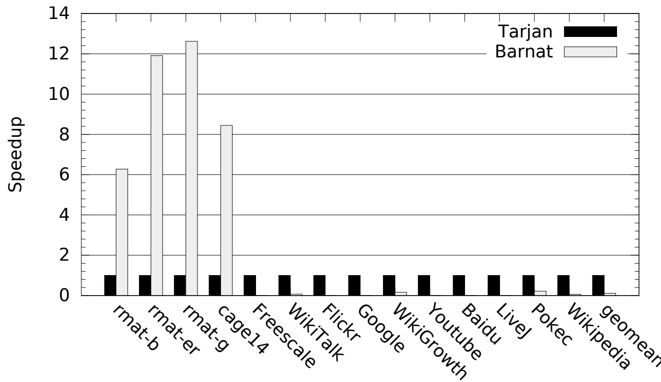

Figure 1: Performance of Barnat's CUDA SCC detection, normalized to the sequential Tarjan's algorithm.

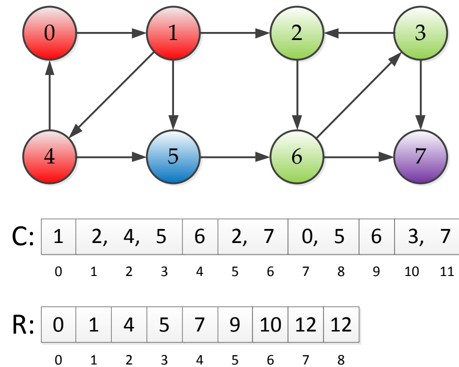

Figure 2: An example of the compressed sparse row (CSR) format. This graph has 4 SCCs (red, green, blue, purple). Note that the blue and purple SCCs are trvial SCCs consisting of only one vertex.

Algorithm 3 Baseline GPU FB-Trim Algorithm

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

indicate whether the corresponding vertex has been visited during the forward and backward traversal. If a vertex is both marked as forward and backward visited, it is identified as an element of the SCC that the pivot belongs to.

Note that we use the well-known compressed sparse row (CSR) sparse matrix format to store the graph in memory consisting of two arrays. Fig. 2 provides a simple example. The column-indices array C is formed from the set of the adjacency lists concatenated into a single array of m (m is the number of edges) integers. The row-offsets array R contains $n + 1$ (n is the number of vertices) integers, and entry $R[i]$ is the index in C of the adjacency list of the vertex v_i.

3.2 Hybrid Method

As mentioned, real-world graphs with a power-law distribution have fundamentally different characteristics compared to traditional artificial graphs, making the existing FW-Trim method extremely inefficient. Previous studies [4, 14, 29] revealed the major characteristic of real-world graphs that mostly affects the performance of SCC detection: the existence of a single giant SCC and a lot of small sized SCCs. Fig. 10 (1) illustrates this characteristic in a real-world graph instance which is the link relationship of a blog sphere named LiveJournal [17]. This characteristic causes load imbalance and serialization problems [14] in the FB-Trim algorithm.

To handle the irregular characteristic, we propose a hybrid method. In this algorithm structure, the SCC detection problem is solved in two phases with different parallelism strategies. During 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 utilize different parallelism strategies customized to the workload characteristics of each phase, maximizing performance for both phases.

Fig. 3 illustrates the execution time distribution of Barnat's CUDA SCC detection. For `rmat` graphs and `cage14` (not shown in the figure), there is only one nontrivial SCC (i.e. the single giant SCC), and thus no Phase-2 is needed for these graphs. For the other graphs, we observe that most of the time is spent on Phase-2 to process the large amount of small sized SCCs. This is because Phase-2 is scarcely parallelized even when a large number of SCCs are identified in this phase. This serialization is due to the fact that *the large amount of remaining small sized subgraphs are disconnected to each other*, and recursively applying the FB algorithm to each subgraph will only identify one SCC to which the pivot belongs, but does not provide further partitioning [14]. Consequently, processing the disconnected subgraphs is almost serialized. With this serialization problem, the FB-Trim algorithm needs thousands of iterations to complete for most of the small-world graphs in our benchmarks (since these graphs have thousands of small sized nontrivial SCCs). Note that although BFS within each subgraph is still parallelized, it can offer very limited parallelism since these subgraphs are small.

Algorithm 4 FB-Trim-Hybrid Algorithm

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

3.3 Exploiting Parallelism in Phase-2

To handle the serialization problem in Phase-2, we apply the extensions that Hong *et al.* [14] proposed in their parallel CPU implementation to our GPU implementation. We refer this implementation as FW-Trim-Hybrid. The extensions that FW-Trim-Hybrid applies to FW-Trim are: 1) finding weakly connected components (`FWCC`), and 2) detecting size-2 SCCs (`Trim2`).

As mentioned in 3.2, after the giant SCC is identified and removed in Phase-1, Phase-2 is mostly serialized because the small sized SCCs are disconnected to each other. 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 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 GPU

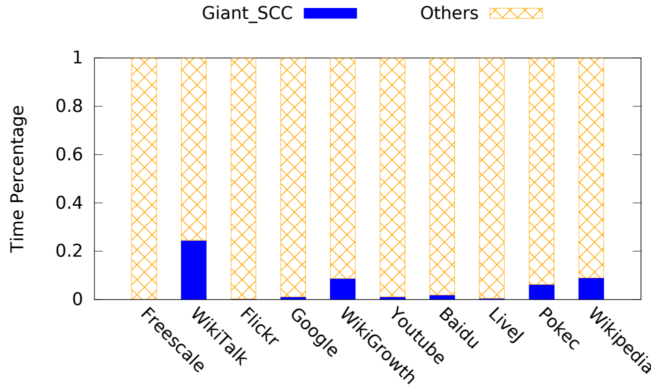

Figure 3: Exection time distribution of Barnat's CUDA implementation. Time spent on processing the giant SCC takes only a small portion of the total time. Most of the time is taken to process the rest small sized SCCs since this phase is scarcely parallelized.

implementations of `FWCC` and `Trim2` are quite straightforward and thus not shown here.

Algorithm 4 shows the algorithm skeleton of FB-Trim-Hybrid. In Phase-1 (line 3~11), the single giant SCC is decomposed. The transition between Phase-1 and Phase-2 occurs when an SCC containing more than 1% of the nodes 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). After that WCC is calculated (line 14) to exploit parallelism. In Phase-2 (line 17~23), a large amount of small sized SCCs are detected. Our experiments in Section 4 show that the FWCC extension can dramatically increase the parallelism of Phase-2, leading to significant execution time reduction. Fig. 7 illustrates the execution time distribution of FB-Trim-Hybrid. Compared to Fig. 3, we observe that the portion of time spent on Phase-1 increase a lot due to the execution time reduction of Phase-2. Next, we try to optimize the major operations in each phase.

3.4 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 [3, 5]. Basically two parallelism strategies are utilized on GPUs: *topology-driven* or *data-driven* implementations [25].

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.* [11] first developed topology-driven BFS on GPUs. Hong *et al.* [13] improved it by mapping warps rather than threads to vertices.

By contrast, the data-driven implementation maintains a worklist which holds the remaining vertices to be processed. In each iteration, threads are created in proportion to the size of the worklist (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 the load imbalance problem, since vertices may have different amount of edges to be processed by the corresponding threads. Merrill *et al.* [23] 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`),

Algorithm 5 Forward-Reach Procedure (topology-driven)

1: **procedure** FWD-REACH($G(V, E)$, *visited*, *color*, *mark*)
2:   **repeat**
3:     $changed \leftarrow false$
4:     **for** each vertex $v \in V$ in `parallel` **do**
5:       **if** $!mark(v)$ and $visitied(v).fw = true$ **then**
6:         FW-STEP(G, v, *visited*, *color*, *mark*, *changed*)
7:       **end if**
8:     **end for**
9:   **until** $changed = false$
10: **end procedure**

Algorithm 6 Forward-Step Kernel (topology-driven)

1: **procedure** FW-STEP(G, v, *visited*, *color*, *mark*, *changed*)
2:   **for** each vertex $w \in adj(v)$ **do**
3:     **if** $!mark(w)$ and $color(w) = color(v)$ **then**
4:       $visited(w).fw \leftarrow true$
5:       $changed \leftarrow true$
6:     **end if**
7:   **end for**
8: **end procedure**

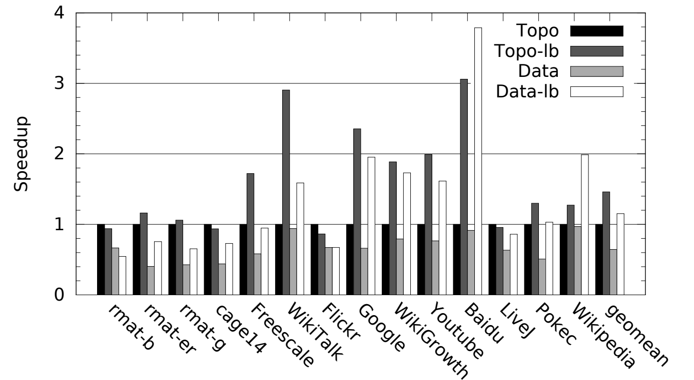

Figure 4: Performance of topology-driven v.s. data-driven implementations when processing the single giant SCC in Phase-1, all normalized to the `topo` implementation.

Algorithm 7 Backward-Reach Procedure (data-driven)

```
1:  procedure Bwd-Reach(G^T(V, E), visited, color, mark)
2:      W_in ← pivots
3:      while W_in ≠ ∅ do
4:          for each vertex v ∈ W_in in parallel do
5:              if !mark(v) and visitied(v).bw = true then
6:                  Bw-Step(G, v, visited, color, mark)
7:              end if
8:          end for
9:          swap(W_in, W_out)                    ▷ Swap the worklists
10:     end while
11: end procedure
```

Algorithm 8 Backward-Step Kernel (data-driven)

```
1: procedure Bw-Step(G, v, visited, color, mark)
2:     for each vertex w ∈ adj(v) do
3:         if !mark(w) 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
```

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 implementation of the Fw-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 Fw-Step kernel operations. In real implementation, a data structure expanded is used to indicate whether the corresponding vertex has been expanded or not during the traversal, so as to filter expanded vertices and remove unnecessary work.

Algorithm 7 shows the naive data-driven implementation of the Bw-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 workers become idle and no work items remain in the worklist. *Double buffering* [25] is used to avoid copying the worklist. Algorithm 8 illustrates the Bw-Step kernel operations.

Fig. 4 and Fig. 5 compare the performance of Phase-1 and Phase-2 using these four BFS implementations respectively. In Fig. 4, we observe that without load balancing, topo consistently outperforms data, since data has extra overhead caused by maintaining the worklist. After applying load balancing, both version get substantial speedup. For some graphs, e.g. Baidu and Wikipedia, data-lb shows significant speedups (even better than topo-lb) due to its work-efficiency. On average topo-lb achieves the best performance among the four, with a geomean speedup of 1.45× over topo.

Fig. 4 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. 5 illustrates that load balancing is not effective since its overhead exceeds its performance benefits, although Wiki-growth and Wiki-pages can still benefit from load balancing and get speedup with data-lb. According to this observation, we decide to first apply topo-lb in Phase-1 and then switch to topo in Phase-2.

3.5 Implementation Details

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. In this paper, we use this method to generate pivots.

Read-only Data Caching. In CUDA devices of compute capability 3.5 and higher, data that is read-only for the entire lifetime of the kernel can be kept in the read-only data (unified L1/texture) cache by reading it using the intrinsic __ldg() [26]. We use the read-only cache to hold the C array and the R array. In this way, we capture temporal locality and improve the performance by reducing the total number of DRAM accesses.

4 EVALUATION

We use the R-MAT [8] graph generator GTGraph [20] to create synthetic graphs. The generator determines the degree distribution by using four non-negative parameters (a; b; c; d) whose sum equals one. We generate three graphs (rmat-er, rmat-g and rmat-b) with 1M vertices size but varying structures by using the following set of parameters: (0:25; 0:25; 0:25; 0:25); (0:45; 0:15; 0:15; 0:25); (0:55; 0:15; 0:15; 0:15). We also pick real-world graphs from the University of Florida Sparse Matrix Collection [1], the SNAP database [17] and

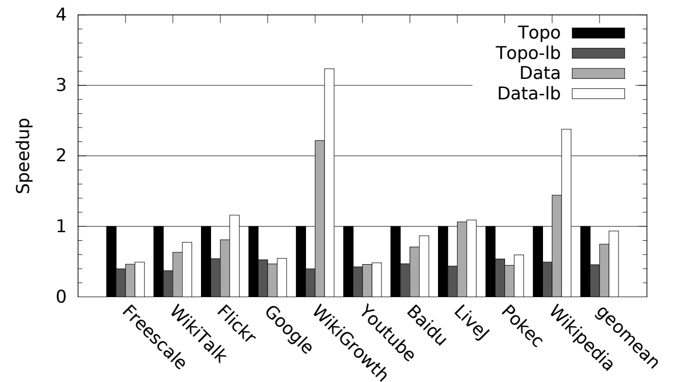

Figure 5: Performance of topology-driven v.s. data-driven implementations when processing the small sized SCCs in Phase-2, all normalized to the topo implementation.

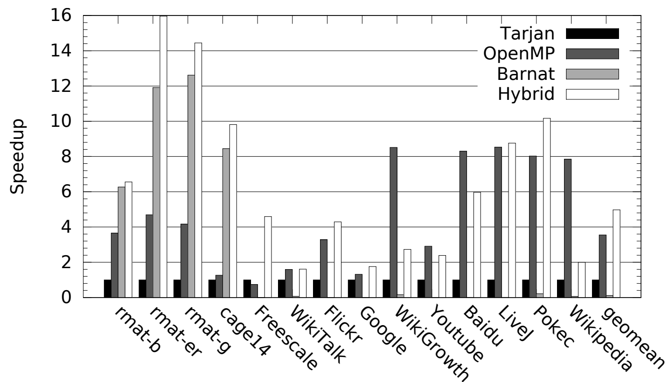

Figure 6: Performance of the SCC detection implementations, all normalized to the sequential Tarjan's algorithm.

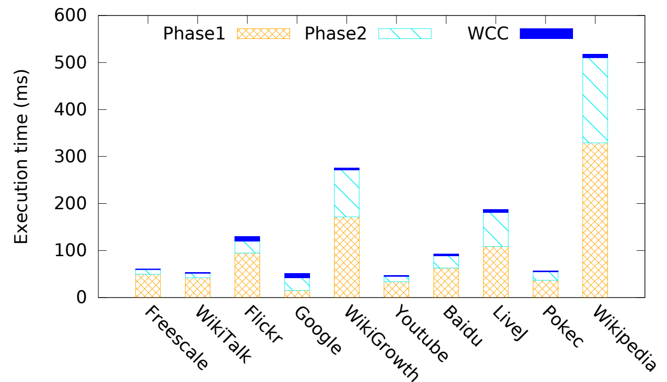

Figure 7: Execution time breakdown of our proposed FB-Trim-Hybrid implementation.

the Koblenz Network Collection [2]. These benchmarks are also used in previous work [14, 29, 30]. The matrices with the respective number of vertices and edges are shown in Table 1. In summary, we use 3 synthetic graphs and 11 real-world graphs for our evaluation. The graphs vary widely in size, degree distribution, density of local subgraphs and application domain.

4.1 Experiment Setup

We compare 4 implementations including (1) `Tarjan`: Tarjan's serial algorithm implemented in [4], (2) `OpenMP`: Hong's OpenMP implementation [14], (3) `Barnat`: Barnat's CUDA implementation [4], (4) `Hybrid`: our proposed GPU implementation FB-Trim-Hybrid. We conduct the experiments on the NVIDIA K20c GPU with CUDA Toolkit 7.5 release. `Tarjan` and `OpenMP` is executed on Intel Xeon E5 2670 2.60 GHz CPU with 8 cores. We launch 16 threads for `OpenMP` since this is the best performing configuration as we 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 Performance

Fig. 6 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 5.0× compared to the Tarjan's serial one, while `OpenMP` gets 3.5× performance improvement. Compared to `OpenMP`, our method is 40% 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. For `Wikipedia` and `WikiGrowth`, `OpenMP` is much faster than our method, and we find that this is due to the fact that BFS on these two graph instances works better on the CPU than on the GPU. It is most likely that the topology of these two graphs affects BFS performance. Note that our BFS implementation uses generic optimization techniques that is portable to various GPU architectures. If extremely optimized for Kepler architecture, graph traversal can be further accelerated with more aggressive optimization techniques. This will be our future work.

As mentioned, `Barnat` get speedup for the first four graphs (because these graphs have only one non-trivial SCC), but it is much slower than `Tarjan` on average. Our method, however, consistently works better than `Tarjan` and `Barnat`, although for some benchmarks, e.g. `Google`, the speedup is very limited due to the graph topology. For `rmat` graphs and `cage14`, `Hybrid` is faster than `Barnat` thanks to the optimized BFS implementation (see details in Section 3.4), while for the rest real-world graphs, our method outperforms `Tarjan` and `Barnat` mainly because of the much higher parallelism exploited by the WCC method. Table 2 shows that WCC substantially reduces the number of iterations required to complete SCC detection. For many real-world graphs, without WCC, `Barnat` needs thousands of iterations to finish since its Phase-2 is almost sequential. By contrast, our method terminates within several or a dozen iterations. In general, our GPU method is more practical and efficient than the existing one.

To better understand the performance effect of our optimization techniques, we breakdown the execution time into three parts: Phase-1, WCC and Phase-2, shown in Fig. 7. As expected, since WCC can exploit parallelism, execution time spent in Phase-2 is significantly reduced so that it doesn't dominate the total execution time any more. Meanwhile, with refined BFS implementation, the Phase-1 performance is also improved. Besides, we parallelize our WCC implementation on the GPU and ensure its low overhead. Orchestrating all the three parts with customized optimization techniques transforms into the final performance speedup.

4.3 Sensitivity to Input Scale

We evaluate the sensitivity of our method when changing the size of the input datasets, shown in Fig. 8. In this experiment, we change the graph size from 1M to 16M vertices, with a fixed density (average degree) of 10. The figure shows the execution time speedup over Tarjan's sequential method. It is clear that our method consistently

Graph	# Vertices	# Edges	Largest SCC Size	Max. deg.	Avg. deg.	Description
rmat-er	1048576	10485760	1047016	49	10.0	Synthetic random graph
rmat-g	1048576	10485760	969560	692	10.0	Synthetic random graph
rmat-b	1048576	10485760	614556	10114	10.0	Synthetic random graph
cage14	1505785	27130349	1505785	41	18.0	DNA electrophoresis
Freescale	2999349	23042677	2888522	30478	7.7	Circuit simulation
WikiTalk	2394385	5021410	111881	3311	2.1	Wikipedia talk (communication) network
Flickr	2302925	33140017	1605184	18022	14.4	Connection of Flickr users
Google	875713	5105039	434818	6326	5.8	Web graph from Google
WikiGrowth	1870709	39953145	1629321	225883	21.4	English Wikipedia with edge arrival times
Youtube	1138499	4942297	509245	25487	4.3	Youtube users and their connections
Baidu	2141300	17794839	609905	97848	8.3	Chinese online encyclopedia Baidu
LiveJ	4847571	68993773	3828682	13906	14.2	LiveJournal online social network
Pokec	1632803	30622564	1304537	13733	18.8	Pokec online social network
Wikipedia	3148440	39383235	2104115	168685	12.5	Links in Wikipedia pages

Table 1: Suite of benchmark graphs

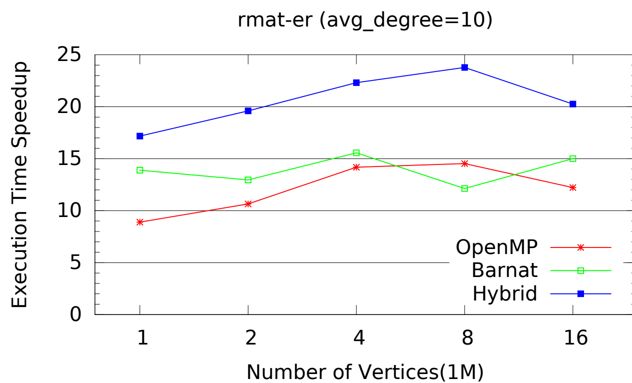

Figure 8: Performance of the SCC detection with varied dataset size (1M to 16M vertices), all normalized to the sequential Tarjan's algorithm.

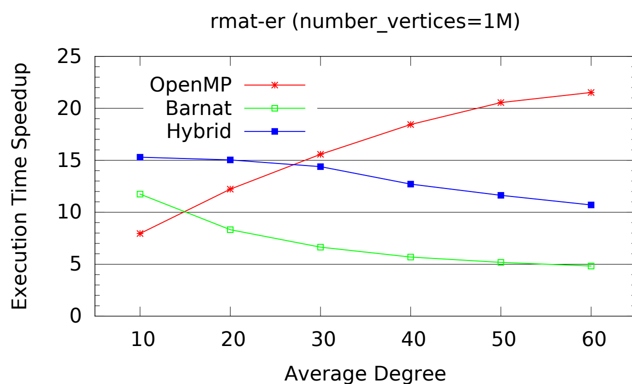

Figure 9: Performance of the SCC detection with varied graph density, all normalized to the sequential Tarjan's algorithm.

outperforms the existing CPU and GPU 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 20× over the Tarjan's algorithm. By contrast, Barnat shows limited performance superiority compared to OpenMP. Note that we currently focus on single-GPU implementation, and thus processing even larger graphs with multi-GPU or multi-node machine will be our future work.

4.4 Sensitivity to Density

To understand the effect of graph density on performance, we conduct another sensitivity study. Fig. 9 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. However, 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

Graphs	Barnat	Hybrid	# nontriv.
rmat-er	1	1	1
rmat-g	1	1	1
rmat-b	1	1	1
cage14	1	1	1
Freescale	55084	2	55084
WikiTalk	456	3	568
Flickr	28804	6	58636
Google	5347	14	12874
WikiGrowth	2702	4	2835
Youtube	10752	6	11370
Baidu	9371	5	22282
LiveJ	12226	5	23456
Pokec	1080	3	2094
Wikipedia	2559	5	2666

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

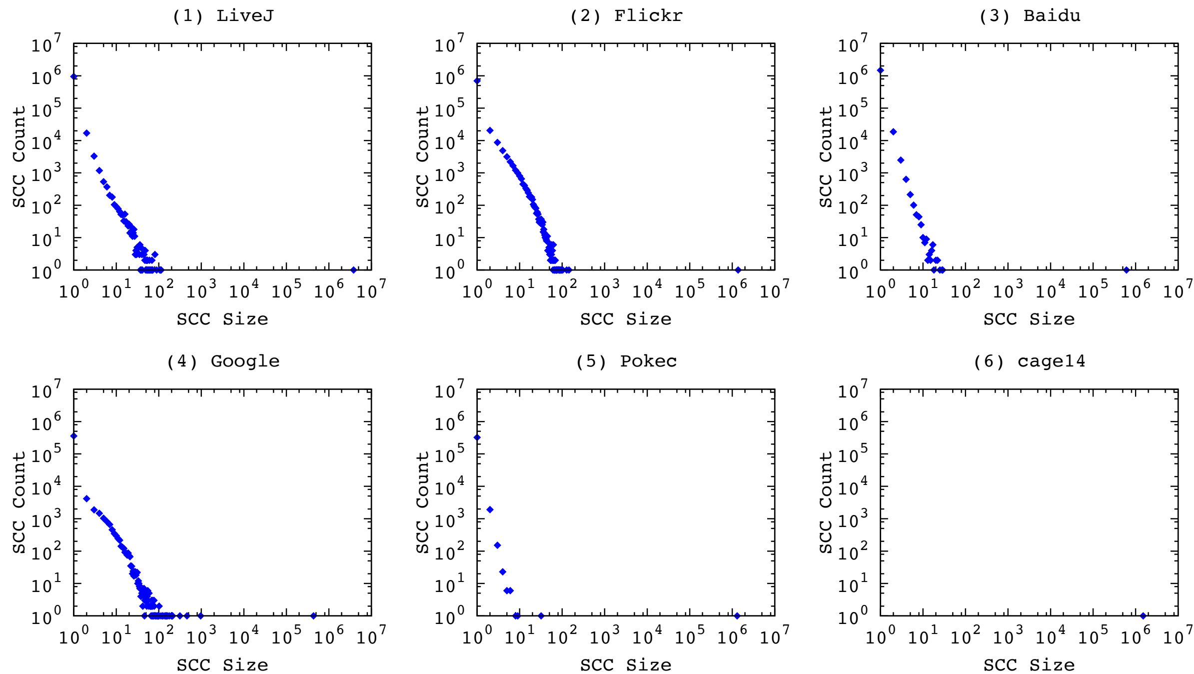

Figure 10: Distribution of SCC sizes of some graph instances that are used in our experiments.

working set size is proportional to the graph density, GPUs may suffer higher degree of memory divergence and cache thrashing with limited on-chip cache size. This observation suggests us to optimize cache behavior 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 therefore our GPU method can achieve better performance than the CPU parallel implementation.

4.5 SCC Distribution

Fig. 10 shows the SCC structure of some graphs used in the experiments. As mentioned, for the small-world graph instances, e.g., the first five graphs in Fig. 10, there is usually a single giant SCC and many small-sized non-trivial SCCs. Although trivial SCCs are the most frequent, they can be efficiently handled by the *Trim* step. For non-small-world graphs, e.g., `cage14`, the SCC related property is quite different. `cage14` has only a single giant SCC, which means its vertices are all strongly connected. In this case, GPU implementations can easily exploit parallelism and achieve good performance as shown in the previous GPU BFS work. Therefore the major part that hurts performance is the large amount of small-sized non-trivial SCCs. Our proposed implementation imports the WCC method to efficiently deal with this case.

5 RELATED WORK

Parallel SCC detection is an important graph analysis algorithm that has been intensively studied previously. Hong *et al.* [14] proposed an efficient parallel CPU SCC detection method specifically for processing real-world graphs. They were the first to use WCC method to handle power-law graphs, and got good speedup and scalability on multicore CPUs. Our work employs the WCC method on GPUs, and further optimizes the algorithm to leverage GPU's compute capability. Slota *et al.* [29] used another strategy to deal with power-law graphs. Their `Multistep` method combines BFS and coloring-based methods and uses them in different algorithm steps. Barnat *et al.* [4] were the first to implement FB-Trim algorithm on GPUs. They used synthetic graph instances as benchmarks, and got dramatic performance speedup. However, their work did not pay specific attention on dealing with the irregularity in real-world graphs. Slota *et al.* [30] implemented their `Multistep` method on GPUs to handle the large amount of small-sized SCCs in real-world graph. Instead of using coloring algorithm, our GPU implementation imports Hong's WCC method to handle this problem. More importantly, our method enables adoption of different graph traversal strategies for different algorithm phases.

Many other graph algorithms have been developed on GPUs. Harish *et al.* [11] are the pioneers to implement GPU graph algorithms. They developed topology-driven Breadth-first Search (BFS) and shortest path algorithms. Hong *et al.* [13] proposed another topology-driven BFS to map warps rather than threads to vertices. Luo *et al.* [19] developed the first data-driven BFS on GPUs. Merrill *et al.* [23] improved Luo's work. They employed 1) prefix sum to reduce atomic operations and 2) dynamic load balancing to deal with scale-free graphs. This implementation achieves high throughput and good scalability. The two major techniques of their work are also applicable to our implementation, while our work focuses more on the algorithm-specific refinement, e.g., the specific strategies to alleviate side effects of GPU's massive

parallelism. Davidson *et al.* [9] developed a work-efficient Single-Source Shortest Path (SSSP) algorithm on GPUs. They improve load balance by partitioning the work into chunks and assigns each chunk to a thread block. These work demonstrated that with careful mapping and optimizations graph algorithms can get substantial performance boost on the GPU. 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.

6 CONCLUSION

SCC detection is an important graph algorithm that has been applied in many application domains. To process large-scale graphs, parallel SCC detection has been intensively studied in the past. Meanwhile, GPUs have been broadly utilized to speed up compute intensive kernels of HPC applications in the last decade. In this paper, we explore the efficient implementation of parallel SCC detection on the GPU. Existing implementations achieve good performance for synthetic graphs but work poorly when applied to real-world graphs. We present a GPU SCC detection implementation that offers high performance for both synthetic and real-world graphs. We propose a hybrid method and customize parallelism strategies for different algorithm phases. We also employ algorithm extensions to handle the irregularity of real-world graphs. Experimental results show that our proposed implementation substantially outperforms existing GPU implementations. This work helps us further understand graph algorithms on modern massively parallel processors, and gives insight on the importance of both algorithm adaptation and architecture-specific optimizations to handle the data irregularity of real-world graphs and fully take advantage of the underlying GPU hardware.

7 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] 2011. The University of Florida Sparse Matrix Collection. (2011). http://www.cise.ufl.edu/research/sparse/matrices/

[2] 2013. Koblenz network collection. (2013). http://konect.uni-koblenz.de

[3] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. 2010. Scalable Graph Exploration on Multicore Processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 1–11.

[4] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. 2011. Computing Strongly Connected Components in Parallel on CUDA. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 544–555.

[5] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Article 12, 10 pages.

[6] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. 2000. Graph Structure in the Web. *Computer Networks* 33, 1-6 (June 2000), 309–320.

[7] M. Burtscher, R. Nasre, and K. Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 141–151.

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *SDM*.

[9] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, 349–359.

[10] Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. 2000. On Identifying Strongly Connected Components in Parallel. In *Proceedings of the 15th IPDPS Workshops*, 505–511.

[11] Pawan Harish and P. J. Narayanan. 2007. *Proceedings of the 14th International Conference High Performance Computing (HiPC).* Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Accelerating Large Graph Algorithms on the GPU Using CUDA, 197–208.

[12] Ramin Hojati, Robert K. Brayton, and Robert P. Kurshan. 1993. BDD-Based Debugging Of Design Using Language Containment and Fair CTL. In *Proceedings of the 5th International Conference on Computer Aided Verification*, 41–58.

[13] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 267–276.

[14] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. 2013. 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)*, Article 92, 11 pages.

[15] Seyed Jalal Kazemitabar and Hamid Beigy. 2009. 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*, 829–834.

[16] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. 2006. Structure and Evolution of Online Social Networks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 611–617.

[17] J. Leskovec. 2013. SNAP: Stanford Network Analysis Platform. (2013). http://snap.stanford.edu/data/index.html

[18] Pingfan Li, Xuhao Chen, Zhe Quan, Jianbin Fang, Huayou Su, Tao Tang, and Canqun Yang. 2016. High Performance Parallel Graph Coloring on GPGPUs. In *Proceedings of the 30th IPDPS Workshop*, 1–10.

[19] Lijuan Luo, Martin Wong, and Wen-mei Hwu. 2010. An Effective GPU Implementation of Breadth-first Search. In *Proceedings of the 47th Design Automation Conference (DAC)*, 52–55.

[20] K. Madduri and D. A. Bader. 2006. GTgraph: A suite of synthetic graph generators. (2006). http://www.cse.psu.edu/fkmadduri/software/GTgraph/

[21] Adam McLaughlin and David A. Bader. 2014. Scalable and High Performance Betweenness Centrality on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 572–583.

[22] William McLendon III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. 2005. Finding Strongly Connected Components in Distributed Graphs. *Journal of Parallel and Distributed Computing (JPDC)* 65, 8 (Aug. 2005), 901–910.

[23] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 117–128.

[24] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, 29–42.

[25] R. Nasre, M. Burtscher, and K. Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *Proceedings of the 27th IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 463–474.

[26] NVIDIA 2015. *CUDA C Programming Guide v7.0.* NVIDIA.

[27] Alex Pothen and Chin-Ju Fan. 1990. Computing the Block Triangular Form of a Sparse Matrix. *ACM Transactions Mathematical Software (TOMS)* 16, 4 (Dec. 1990), 303–324.

[28] John H. Reif. 1985. Depth-first search is inherently sequential. *Inform. Process. Lett.* 20, 5 (1985), 229 – 234.

[29] G. M. Slota, S. Rajamanickam, and K. Madduri. 2014. 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)*, 550–559.

[30] G. M. Slota, S. Rajamanickam, and K. Madduri. 2015. High-Performance Graph Analytics on Manycore Processors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 17–27.

[31] G. M. Slota, S. Rajamanickam, and K. Madduri. 2016. Parallel Graph Coloring for Manycore Architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 1–10.

[32] Bc Miroslav Stuhl. 2013. Computing Strongly Connected Components with CUDA. (2013).

[33] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[34] Aiguo Xie and P. A. Beerel. 2006. Implicit Enumeration of Strongly Connected Components and an Application to Formal Verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 19, 10 (Nov. 2006), 1225–1230.