# External GPU Biconnected Components

Abhijeet Sahu[1], Andaluri S P V M Aditya[1], G Ramakrishna[1], Malleti Sai Nikhil[1], Kishore Kothapalli[2], and Dip Sankar Banerjee[3]

[1] IIT Tirupati, Tirupati, India
cs22s501@iittp.ac.in, cs21b002@iittp.ac.in, rama@iittp.ac.in, cs21m009@iittp.ac.in
[2] IIIT Hyderabad, Hyderabad, India
kkishore@iiith.ac.in
[3] IIT Jodhpur, Jodhpur, India
dipsankarb@iitj.ac.in

**Abstract.** As the scale of graph analytics continues to grow, many applications require identifying biconnected components (BCCs) and cut vertices in graphs that exceed the memory capacity of a single GPU. This paper presents an out-of-core, GPU-based batch processing algorithm designed to efficiently compute BCCs and cut vertices in massive graphs that do not fit entirely into device memory. We propose a novel batch technique to process the graph incrementally, and maintain a Biconnectivity Compressed Graph to compute BCCs and cut vertices. Experimental results on a range of large-scale benchmark graphs demonstrate that our technique achieves competitive performance compared to state-of-the-art CPU solutions, enabling the handling of graph instances previously considered intractable on GPU platforms.

**Keywords:** Large-scale graphs, Biconnected components, Articulation points, Cut vertices, Out-of-core processing, GPU, Batch processing

## 1 Introduction

Processing large-scale graphs is increasingly vital across many fields as both the complexity and size of networked data continue to grow. A recent survey [14] reports that many organizations now manage graphs containing billions of edges and requiring hundreds of gigabytes of storage, highlighting the need for scalable parallel approaches. Although parallel architectures (e.g., multi-core CPUs and GPUs) have driven advancements in graph analytics, the limited on-board memory of GPUs [1] remains a major obstacle for handling large-scale graph data. Indeed, the most pressing challenge in GPU-based graph applications, according to [14] is performing computations on these massive graphs.

Among various graph-theoretic tasks, the identification of *cut vertices* (articulation points) and *biconnected components* (BCCs) is especially important. Cut vertices reveal critical junctions in networks (e.g., for routing or vulnerability

[1] For example, NVIDIA A100 GPUs typically have only 40 GB of on-board memory.

analysis), whereas BCCs denote maximal subgraphs that remain connected after removing any single vertex. These properties underpin a range of higher-level algorithms, including partitioning, centrality computations and planarity testing [6]. As large-scale graph processing frequently employs divide-and-conquer strategies, designing efficient GPU-based methods to identify these components remains a key challenge.

Although biconnectivity is a classical problem solved in theory over half a century ago, it continues to attract interest in modern parallel contexts. Wadwekar and Kothapalli proposed the first GPU-based parallel algorithm for BCC [18], constructing a breadth-first search (BFS) tree, then using fundamental cycle traversals of non-tree edges to build an auxiliary graph whose connected components directly yield the BCCs.

Beyond this early GPU-focused effort, various approaches exist across different architectures. In sequential settings, the Hopcroft-Tarjan (HT) algorithm [8] runs in linear time via depth-first search (DFS) to identify BCCs. Tarjan and Vishkin (TV) [17] introduced a classic parallel method that replaces DFS with an arbitrary spanning tree (AST), transforming the input graph $G$ into a skeleton graph $G'$. Although elegant, this approach expands the vertex set to $O(m)$ (each edge in $G$ becomes a vertex in $G'$), leading to substantial memory overhead.

To reduce the size of $G'$, Cong and Bader [3] proposed the *TV-filter*. They construct a BFS tree $T$ of $G$ and a spanning forest $F$ of $G - T$. Any edge not in $T \cup F$ is deemed "non-essential" for biconnectivity, thereby reducing $|V(G')|$. Although BFS-based approaches work well in standard parallel or GPU environments, BFS traversals can be challenging to implement efficiently in an external setting. More recently, Dong et al. [5] introduced *FAST-BCC*, now considered the state of the art for multicore implementations. Their algorithm identifies the BCCs of $G$ by finding connected components of a subgraph $G' \subseteq G$, comprised of non-critical edges in a spanning tree $T$ plus cross edges in $G - T$.

All of these methods implicitly assume that the full graph (all edges along with extra auxiliary data structures) fits into GPU memory. In contrast, we consider a scenario in which the GPU can store all vertices (i.e., $\mathcal{O}(n)$ space) but does not have enough memory for the complete set of edges, which may be significantly larger. We refer to this GPU-EXTERNAL-MEMORY (GEM) model, wherein the limited onboard memory in GPU cannot accommodate all edges. In this work, we address the *out-of-memory* setting for GPUs by proposing a new algorithm that efficiently computes BCCs on large graphs whose edge set exceeds GPU memory capacity while retaining the GPUs computational advantages.

We use $G = (V, E)$ to denote a connected and undirected graph with $|V| = n$ vertices and $|E| = m$ edges. A *cut vertex* in $G$ is a vertex whose removal (along with its incident edges) results in a disconnected graph. A maximal subgraph of $G$ without any cut vertex is called a *biconnected* component. Given a graph $G$, the *Biconnected Components* (BCC) problem seeks to identify all the biconnected components of $G$.

### 1.1 Our contributions

From the preceding discussion, it is evident that designing an BCC algorithm in GEM model to handle massive graphs remains a challenging problem. Our key contributions are:

- We design a novel algorithm to solve BCC in GEM model. Our algorithm maintains a compressed graph $H$ of $G$ in GPU, where multiple vertices of $G$ are merged into a single vertex in $H$, $size(H) = O(n)$, and BCC($G$) can be obtained from $H$, where $n := |V(G)|$. To the best of our knowledge, this is the first parallel space-efficient out-of-memory BCC algorithm capable of handling massive graphs that exceed GPU memory capacity in practice.
- We introduce *biconnectivity compressed graph* (BCG) of a graph along with necessary properties. These properties help to obtain BCCs of the input graph.
- We provide exhaustive proofs to support the correctness of the proposed algorithm, ensuring both theoretical rigor and practical applicability.
- We evaluate our algorithm on large, real-world datasets, demonstrating that it outperforms state-of-the-art approaches in efficiency and scalability.

### 1.2 Related Work

Multiple graph management frameworks have evolved to move data back and forth between CPU and GPU [13], [16], [11] to deal with the out-of-memory issues in GPU. The Subway framework loads all the active graph edges to GPU [13] in every iteration of a graph algorithm. Ascetic and Liberator frameworks only load active graph edges that do not appear in GPU, using unified virtual memory and zero-copy mechanism [16][11]. These frameworks help programmers use existing PRAM-based graph algorithms and implementations and increase productivity. However, there is no bound on the number of times graph data loaded from CPU to GPU, as PRAM-based algorithms are borrowed to GEM model. Inspired by functional programming, the Graph-Reduce framework uses the Gather-Apply-Scatter (GAS) method to handle graphs that do not fit into GPU [15]. GAS paradigm may not be directly applicable for graph problems such as BCC. Even so, these frameworks do not address the question of the number of passes of the input needed by a given computation. This poses questions on their efficiency. In our work, we design a new algorithm to solve BCC suitable for GEM model, in which we load all the edges of $G$ from CPU to GPU precisely twice in the entire algorithm. In particular, our algorithm uses $O(n)$ space in GPU.

## 2 Preliminaries and Biconnectivity Compressed Graph

In this section, we describe two ways of representing important details of biconnected components in a graph, namely *implicit-bcc* labels and *labels and component head* (LCH) representations, along with a brief description about the state-of-the-art multicore algorithm for BCC [5]. Later, we define *biconnectivity compressed graph* (BCG) of a graph along with necessary properties.

**Implicit-BCC Labels.** The labels assigned to vertices of a graph are called *implicit-BCC* labels if they satisfy the following two conditions: 1) Two non-cut vertices are assigned the same label if and only if they belong to the same BCC 2) The label of a cut vertex is distinct from the labels of all other vertices. For a graph $G$, we use an array $I[\cdot]$ to store the *implicit-BCC* labels of $G$, and an example is shown in Fig. 1.

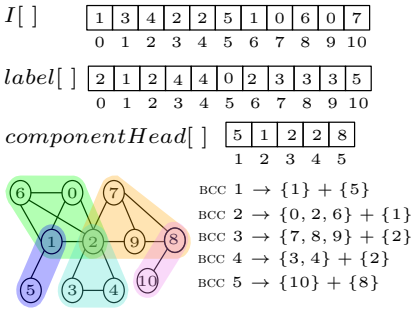


Fig. 1: *Implicit-BCC* array and LCH.

**LCH Representation for BCCs.** For a connected graph $G$, the Label and ComponentHead (LCH) representation is defined with respect to a spanning tree of $G$ rooted at $r$. The $label[\cdot]$ array contains labels for all vertices expect $r$, representing the BCC to which each vertex belongs. Each distinct label is associated with a component head vertex. It is important to note that all vertices sharing the same label, along with its component head, form a BCC in $G$. An example of this representation is shown in Fig. 1 where $r = 5$. More details about this representation are discussed in [2][5].

**FAST-BCC Algorithm.** For a connected graph $G$ as input, this algorithm ([5]) begins by constructing a rooted spanning tree $T$ of $G$. A tree edge $(parent[u], u)$ is referred to as *fence* edge if no nontree edge from $u's$ subtree escapes from $parent[u]'s$ subtree. Tree edges that are not fence are called *plain* edges. A skeleton graph $G'$ is constructed with plain edges in $T$ and all cross edges of $G$ with respect to $T$. BCCs information of $G$ is obtained from the connected components of $G'$ in the LCH format.

**Biconnectivity Compressed Graph (bcg).** Let $G$ and $H$ be two graphs, and let $f : V(G) \to V(H)$ be an onto function. The graph $H$ is referred to as a *biconnectivity compressed graph* (BCG) of $G$ with mapping $f$, if $f$ satisfies the following four properties.

**Property 1.** If $u$ is a cut vertex in $G$ then $f(u) \neq f(v)$ for any $v \in V(G)$ such that $v \neq u$.

**Property 2.** For any two distinct noncut vertices $u, v$ from different biconnected components of $G$, $f(u) \neq f(v)$.

**Property 3.** Let $u$, $v$, and $w$ be three distinct vertices in $G$ and $u'$, $v'$, and $w'$ are the vertices in $H$, where $f(u) = u'$, $f(v) = v'$, and $f(w) = w'$. A cut vertex $w$ separates $u$ and $v$ in $G$ if and only if $w'$ separates $u'$ and $v'$ in $H$, and $w'$ is a real cut vertex. A cut vertex $x$ in $H$ is referred to as ***real*** if $|f^{-1}(x)| = 1$.

**Property 4.** For every edge $e(u', v')$ in $H$ there exists an edge between $(u, v)$ in $G$, where $u \in f^{-1}(u')$ and $v \in f^{-1}(v')$.

We have three types of vertices in a BCG $H$ of $G$, namely *real cut vertices*, *false cut vertices*, and *block vertices*. A cut vertex that is not real is called a ***false cut vertex***, and a noncut vertex is referred to as a ***block vertex***. In Fig. 2, for a graph $G$ shown in (a), a BCG $H_2$ with mapping function $M_2$ are shown in (e).

In $H_2$, the vertices $\{0, 2, 4, 8\}$ are real cut vertices, 6 is a false cut vertex, and $\{1, 3, 5, 7, 9, 10\}$ are block vertices.

## 3 GEM-BCC Algorithm

We begin this section with the necessary notation and terminology. Later, we elaborate on the purpose and details of various subroutines that will be used in the main algorithm. Finally, we present our GEM-BCC algorithm.

**Notation.** The edges of an input graph $G = (V(G), E(G))$ appear as a sequence $(e_1, e_2, \ldots, e_m)$ of edges in the edge stream format, and need not fit into the GPU memory at once. So, $E(G)$ is divided into $k$ consecutive batches $B_1, B_2, \ldots, B_k$, such that each $B_i$ fits in GPU, where each $B_i$ denotes the $i$-th batch of edges, and $k = \lceil |E|/b \rceil$. We use $G_0$ to denote a spanning tree of $G$ and $H_0$ to denote the same spanning tree. For each $1 \leq i \leq k$, we define $G_i$ as $G_{i-1} + B_i$. For each $1 \leq i \leq k$, we use $H_i$ and $M$ to denote a BCG of $G_i$ and the corresponding mapping function from $V(G)$ to $V(H_i)$, respectively. Suppose $G_{i-1}$ is disconnected, a non-cut vertex in $G_{i-1}$ can become a cut vertex in $G_i$. To avoid this, we consider $G_0$ as a spanning tree of $G$ and ensure that for each $0 \leq i \leq k$, $G_i$ is connected.

**High-Level Overview of GEM-BCC Algorithm.** Our GEM-BCC algorithm runs in two passes. In the first pass, all batches of edges are processed to construct a spanning tree of $G$ and initialize a compressed graph in GPU, with the spanning tree, to ensure that the compressed graph is always connected. In the second pass, we consider each batch of edges at a time, expand the existing compressed graph by including them, and compress it again using an in-memory GPU algorithm for BCC. After the second pass is over, we obtain BCCs of $G$ in LCH representation from the BCCs of the repaired compressed graph. Later, we prove in Lemma 6 that the compressed graph in GPU is a BCG, and thus Theorem 1 ensures the proof of correctness of our GEM-BCC algorithm.

**Challenges.** To obtain a concrete algorithm from the high-level overview, we work on the following key challenges.

1. After including a new batch of edges to the existing compressed graph maintained in GPU, what vertices in the current graph need to be merged to compress further?
2. How do we identify certain non-essential edges from the current compressed graph and ensure that the size of the updated compressed graph is bounded by $O(n)$?
3. How do we maintain a mapping function from the vertices in the original graph to the vertices in the compressed graph, so that BCC of $G$ can be obtained back from the compressed graph?

### 3.1 Subroutines

`GEM-spanning-tree(G)`: This subroutine constructs a spanning tree of the given graph $G$. The full edge stream $E = (e_1, e_2, \ldots, e_m)$ of edges need not fit into the

GPU memory at once. So, we divide $E$ into batches $B_1, B_2, \ldots, B_k$, such that each $B_i$ fits in GPU. This subroutine starts with an empty spanning forest. During $i^{th}$ iteration ($1 \leq i \leq k$), we apply GPU-based parallel incremental algorithm [7], to update the existing spanning forest by processing a batch $B_i$ of edges. The computational depth of this subroutine is $O(k \log^2 n)$, with a total work complexity of $O(kn + m) \log n$.

**`Transform(B, M)`**: This subroutine takes a batch $B$ of edges and a mapping function $M$ as input. For each edge $(u, v)$ in $B$, we obtain the transformed edge $(M(u), M(v))$ in parallel and then remove self-loops and duplicate edges by applying parallel compaction [9]. The computational depth of this subroutine is $O(\log |B|)$, with a total work complexity of $O(|B|)$.

**`Compress-Graph(G)`**: This subroutine is meant for merging all the non-cut vertices in a BCC into a single vertex while retaining cut vertices as it is. To accomplish this, we first run GPU version of FAST-BCC algorithm ([5]) on $G$ to obtain the IMPLICIT-BCC array $I[\cdot]$, and then invoke TRANSFORM$(E(G), I)$. The depth of this procedure is bounded by $O(\log^2 n)$.

**`BCC-Repair(H, M)`**: This procedure receives a compressed graph $H$ of $G$ and a mapping array that maps $V(G)$ to $V(H)$. We apply a GPU-based FAST-BCC algorithm ([5]) on $H$ and treat false cut vertices as non-cut vertices to obtain labels for $V(H)$ and component heads for labels. The graph induced on the vertices with the same label and its component head is a maximal subgraph without a real cut vertex. In our GPU version of the FAST-BCC algorithm, we include fence edges $(parent[u], u)$ in the skeleton graph if $parent[u]$ is a false cut vertex, in accordance with the requirements for this subroutine. The depth of this procedure is bounded by $O(\log^2 n)$.

### 3.2 Algorithm

Now, we shall describe Algorithm 1. Initially, the input graph $G$ appears in CPU in the edge stream format. Our GEM-BCC algorithm proceeds in two passes. In the *first pass*, we construct a spanning tree $H_0$ of $G$ using GEM-SPANNING-TREE subroutine and store $H_0$ in GPU.

In the *second pass*, we partition the edge stream $(e_1, \ldots, e_m)$ of $G$ in CPU into $k$ consecutive batches, denoted as $B_1, B_2, \ldots, B_k$, ensuring that each batch individually fits to GPU memory. As we go forward, we merge multiple vertices into a single vertex in GPU. To precisely know which set of vertices are merged into which vertex, we maintain a mapping function represented using an array $M[\cdot]$. At the beginning, for each vertex $v \in V(G)$, we initialize $M(v) = v$ in parallel in Line 2. During the $i^{th}$-iteration of Algorithm 1, we copy a batch $B_i$ of edges from CPU to GPU and perform the following operations in GPU. We first apply the mapping $M$ to the edges in $B_i$ and include the transformed edges in the current compressed graph $H_{i-1}$, using the subroutine TRANSFORM. Due to the inclusion of a new batch of edges, the number of edges in $H_{i-1}$ is increased. We now obtain an IMPLICIT-BCC $I[\cdot]$ of $H_{i-1}$ and compress $H_{i-1}$ to $H_i$ by using the subroutine COMPRESS-GRAPH. It can be observed that $M$ and $I$ are mapping functions from $V(G)$ to $V(H_{i-1})$ and $V(H_{i-1})$ to $V(H_i)$, respectively.

**Algorithm 1:** GEM-BCC Algorithm

**Input:** A Graph $G(V, E)$ in edge stream format, batch size $B$
**Output:** BCC labels for edges and cut vertex status for vertices in $G$.

$H_0 \leftarrow$ GEM-SPANNING-TREE$(G)$
**for** each vertex $v$ **do in parallel** $M[v] = v$
**for** $i = 1$ *to* $k$ **do**
  $H_{i-1} \leftarrow H_{i-1} +$ TRANSFORM$(B_i, M)$
  $(H_i, I) \leftarrow$ COMPRESS-GRAPH$(H_{i-1})$
  **for** each vertex $u \in V(G)$ **do in parallel** $M[u] \leftarrow I[M[u]]$
  $H_i \leftarrow$ TV-FILTER$(H_i)$
$(l[\cdot], ch[\cdot]) \leftarrow$ BCC-REPAIR$(H_k, M)$
**for** *each vertex* $v \in V(G)$ **do in parallel**
  $label[v] \leftarrow l[M[v]]$
**for** *each distinct label* $j$ **do in parallel**
  $componentHead[j] \leftarrow M^{-1}(ch[j])$

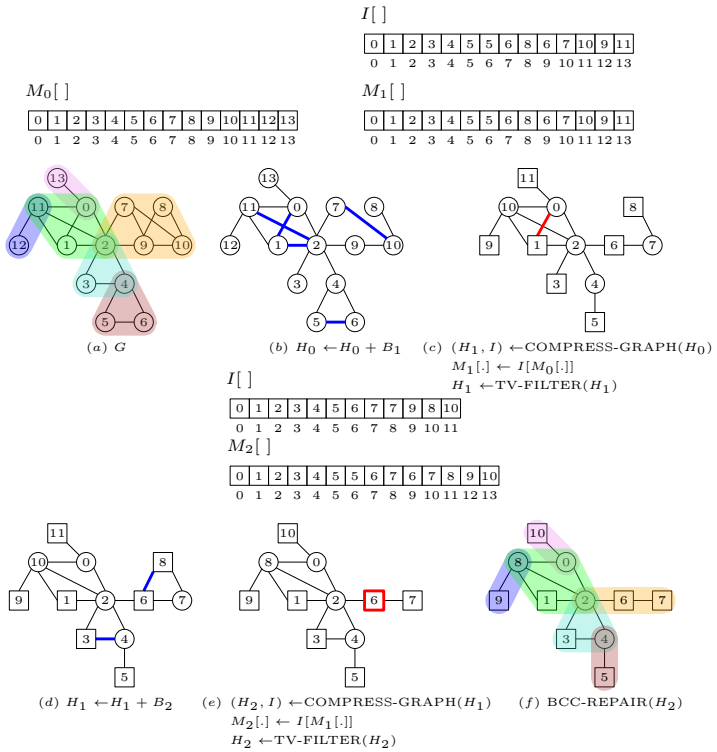


Fig. 2: Illustration of Algorithm 1. (a) An input graph $G$ and BCCs of $G$. (b) $B_1$ edges are shown in blue (c) Edges in red color represent the non-essentail edges to be removed by TV-FILTER (d) $B_2$ edges are shown in blue (e) False cut vertex is shown in red color (f) False cut vertices are treated as non-cut vertices and hence vertices 2, 6, and 7 appear in a single set.

We perform function composition $I \circ M$ to update $M$ in Line 6 so that the new function maps from $V(G)$ to $V(H_i)$. At this stage, $|E(H_i)|$ need not be bounded by $O(|V(H_i)|)$. To achieve this, we eliminate non-essential edges in $H_i$ by exposing it to TV-FILTER algorithm.

After processing all $k$ batches, we are ready with the compressed graph $H_k$. Due to the appearance of fake cut vertices, there is no one-to-one mapping between BCC($G$) and BCC($H_k$). We now apply BCC-REPAIR on $H_k$ to obtain labels to $V(H_k)$ and a component head to each label, such that $(V_1, \ldots, V_r)$ is a vertex decomposition of $V(H_k)$, where $V_i$ denotes a set of vertices with a same label along with its component head and $H[V_i]$ does not have a real cut vertex.

Finally, we obtain labels of $V(G)$ and their component heads from the labels of $H_k$ and their component heads using the mapping function $M$ in Lines 9-12. For a vertex $u$ in $H_k$, note that $|M^{-1}(u)| = 1$, when $u$ is a component head, because every component head is a real cut vertex. The correctness of this algorithm is formally proved in Theorem 1. An illustration of this algorithm is shown in Fig. 2. The total depth of Algorithm 1 is bounded by $O(k(diameter(H_1) + \log^2 n))$ and total work is bounded by $O((kn + m) \log n)$. For each $1 \leq i \leq k$, $|E(H_i)| \leq |V(H_i)| \leq n$, and $|B_i|$ is $O(n)$, the space requirement of this GEM-BCC algorithm in GPU is $O(n)$.

**Obtaining Cut Vertices.** From Property 3 of BCG, there is a one-to-one mapping between cut vertices in $G$ and real cut vertices in $H_k$. All cut vertices in $H_k$ can be retrieved by applying GPU variant of Fast-BCC algorithm [5], and then identify the real cut vertices using the mapping function $M$. The inverse maps of these real cut vertices correspond to the cut vertices in $G$.

## 4 Correctness of External-BCC Algorithm

There are three main operations performed during every iteration of our algorithm. In this section, we prove that after every operation, the resultant compressed graph is a BCG. Later, we show the proof of correctness of the entire algorithm in Theorem 1. Complete proofs can be found at GitHub.

**Lemma 1.** *Let $G$ be a connected graph. Let $H$ be a BCG of $G$ with a mapping function $f : V(G) \to V(H)$. Suppose we add a batch $B$ of edges to $G$ and let $B'$ be the transformed edges of $B$ by applying $f$. Then, $H + B'$ is a BCG of $G + B$, and $f$ remains the associated mapping function.*

*Proof.* Let $w$ be a cut vertex in $G + B$, which separates vertices $u$ and $v$. $w$ is also a cut vertex in $G$ and continues to separate $u$ and $v$, as removing edges while preserving the connectivity of the graph does not alter the property of $w$ as a cut vertex. By Property 3, $f(w)$ is a real cut vertex in $H$, which we denote as $w'$.

By Property 3, for every component, let $C$, in $G - w$ there exists a unique component, we call it $C'$, in $H - w'$ such that a vertex $p$ belongs to $C$ if and only if vertex $f(p)$ belongs to $C'$. Similarly, for every component $C'$ in $H - w'$, there exists a unique component $C$ in $G - w$ such that a vertex $p'$ belongs to $C'$ if

and only if all the vertices in $f^{-1}(p')$ belong to $C$. This is a one-to-one relation between the components in $G - w$ and $H - w'$.

We now denote $i^{th}$ component of $G - w$ as $C_i$, and we denote the component in $H - w'$ corresponding to $C_i$ as $C'_i$. Let $u' = f(u)$ and $v' = f(v)$. Let $u \in C_i$ and $v \in C_j$, where $i \neq j$, which implies $u' \in C'_i$ and $v' \in C'_j$.

If an edge in $B$ is between $w$ and a vertex in component $C_i$, the corresponding edge in $B'$ is between $w'$ and a vertex in component $C'_i$. If an edge in $B$ is between two vertices in the same component $C_i$, then the corresponding edge is between two vertices in $C'_i$. If an edge in $B$ is between vertices from different components $C_i$ and $C_j$, then the corresponding edge in $B'$ is between two vertices from different components $C'_i$ and $C'_j$.

So we conclude edges in $B$ do not create a path between two vertices $u$ and $v$ from different components in $G - w$ if and only if edges in $B'$ do not create a path between $u'$ and $v'$ from different components in $H - w'$. This statement, along with Property 3 of $H$, proves Property 3 for $H + B'$.

As $f$ is not changed and $w$ is the only vertex in $G + B$ which maps to $w'$ in $H + B'$, $f$ satisfies Property 1.

The vertices $u$ and $v$ are in two different biconnected components, and $u'$ and $v'$ are not equal since they are from two different biconnected components in $H + B'$, so $f$ satisfies Property 2. Edges in batch $B'$ are transformed edges of $B$, and the edges of B appear in $G + B$. Hence, Property 4 is satisfied.

We have shown that $H + B'$ satisfies Properties 1, 2, 3, and 4. Therefore, $H + B'$ is a BCG of $G + B$ with respect to the mapping function $f$.

**Lemma 2.** *Let $G$ be a graph. Let $H$ and $f : V(G) \to V(H)$ be compressed graph and mapping function returned by* COMPRESS-GRAPH *subroutine applied on $G$. Then, $H$ is a* BCG *of $G$ with mapping function $f$.*

*Proof.* We know *Implicit-BCC* labels represented by $f$ satisfy following properties:

i. If $u$ is a cut vertex in $G$ then $f(u) \neq f(v)$ for any $v \in V(G)$ such that $v \neq u$. Hence Property 1 is satisfied.
ii. Two noncut vertices are given the same label if and only if they belong to the same biconnected component. Hence, Property 2 is satisfied.
iii. Due to the way of constructing $H$, $E(H) = \{(f(u), f(v)) \mid (u, v) \in E(G), f(u) \neq f(v)\}$. Hence, Property 4 is satisfied.

We now prove that $f$ satisfies Property 3, to declare that $H$ is a BCG of $G$. Let $u$, $v$, and $w$ be three distinct vertices in $G$ and $u'$, $v'$, and $w'$ are the vertices in $H$, where $f(u) = u'$, $f(v) = v'$, and $f(w) = w'$.

We first consider the forward case, in which the cut vertex $w$ separates $u$ and $v$ in $G$. Let $m$ denote the number of connected components in $G - w$. Then, $u$ and $v$ reside in two different components. For each $1 \leq i \leq m$, let $S_i$ denote the set of vertices in $i^{th}$ component. We have the following observations:
Obs 1. $\bigcup_{i=1}^{m} S_i = V(G) \setminus \{w\}$
Obs 2. $S_i \cap S_j = \emptyset$ for $i \neq j$

Obs 3. For any two vertices $p$ and $q$ such that $p \in S_i$, $q \in S_j$, and $i \neq j$, $(p, q) \notin E(G)$.
For each $1 \leq i \leq m$, we define $S'_i = \{f(v) \mid v \in S_i\}$. By Property 1 of BCG, only $w$ maps to $w'$, and by Obs 1, we can infer that $\bigcup_{i=1}^{m} S'_i = V(H) \setminus \{w'\}$. Let $p$ and $q$ be vertices in $S_i$ and $S_j$ respectively, where $i \neq j$. Then $p$ and $q$ do not belong to the same biconnected component in $G$ since all paths between them pass through $w$. By Property P2, $f(p) \neq f(q)$. This fact along with Obs 2 follows that $S'_i \cap S'_j = \emptyset$. From Obs 3, there is no edge connecting any vertex in $S_i$ to any vertex in $S_j$ in $G$. By Property 4, there is no edge connecting any vertex in $S'_i$ to any vertex in $S'_j$ in $H$. Therefore, the subgraph $H[S'_1 \cup \ldots \cup S'_m]$ is disconnected while $H[S'_1 \cup \ldots \cup S'_m \cup \{w'\}]$ is connected, because $V(H) = S'_1 \cup \ldots \cup S'_m \cup \{w'\}$. This implies that $u'$ and $v'$ can only reach each other through $w'$ in $H$. In other words, $w'$ is a cut vertex in $H$ that separates $u'$ and $v'$. Furthermore, due to Property 1, only one vertex $w$ maps to $w'$, and thus $w'$ is a real cut vertex in $H$ and separates $u'$ and $v'$.

Now, we examine the other direction in which the real cut vertex $w'$ separates $u'$ and $v'$ in $H$. Let $m$ denote the number of connected components in $H - w'$. Then $u'$ and $v'$ reside in two different components. For each $1 \leq i \leq m$, let $S'_i$ denote the set of vertices of $H$ in $i^{th}$ component. We have the following observations:
Obs 4. $\bigcup_{i=1}^{m} S'_i = V(H) \setminus \{w'\}$
Obs 5. $S'_i \cap S'_j = \emptyset$ for $i \neq j$
Obs 6. For any two vertices $p'$ and $q'$ such that $p' \in S'_i$, $q' \in S'_j$, and $i \neq j$, $(p', q') \notin E(G)$.
For each $1 \leq i \leq m$, we define $S_i = \{f^{-1}(v) \mid v \in S'_i\}$. By the definition of real cut vertex, only $w$ maps to $w'$. By Obs 4 and the fact that $f$ is well defined for every vertex in $G$, $\bigcup_{i=1}^{m} S_i = V(G) \setminus \{w\}$. Since $f$ is a function, each vertex in $V(G)$ has unique image in $V(H)$. This fact along with Obs 5 follows that $S_i \cap S_j = \phi$ where $i \neq j$. Let $p'$ and $q'$ be vertices in $S'_i$ and $S'_j$ respectively, where $i \neq j$. From Obs 6, no edge exists between $p'$ and $q'$ in $H$. Due to construction procedure of $H$ using IMPLICIT-BCC, no edge between $p'$ and $q'$ implies no edge between any vertex $u \in f^{-1}(p')$ and any vertex $v \in f^{-1}(q')$. So there are no edges between vertices in $S_i$ and $S_j$ in $G$. The subgraph $G[S_1 \cup \ldots \cup S_m]$ is disconnected while $G[S_1 \cup \ldots \cup S_m \cup \{w\}]$ is connected, because $V(G) = S_1 \cup \ldots \cup S_m \cup \{w\}$. Consequently, $w$ is a cut vertex in $G$ and separates $u$ and $v$, and thus Property 3 is satisfied. Therefore $H$ is a BCG of $G$.

**Lemma 3.** *Let $G$ be a connected graph. Let $H$ be a BCG of $G$ with a mapping function $f : V(G) \to V(H)$. Let $H'$ be a graph obtained by TV-FILTER($H$). Then, $H'$ is a BCG of $G$, and $f$ remains the associated mapping function.*

*Proof.* We have the following observations:

i. We know TV-FILTER subroutine does not change the mapping function $f$. Hence, $f$ satisfies Properties 1 and 2 for $H'$.
ii. We know a real cut vertex is also a cut vertex. We also know TV-FILTER subroutine does not change $f$ and also does not change the connectivity of

$H$. If two vertices $u$ and $v$ are separated by a cut vertex $w$ in $H$, then $u$ and $v$ are also separated by cut vertex $w$ in $H'$. Real cut vertices, cut vertices, and non-cut vertices in $H$ remain to act as real cut vertices, cut vertices, and non-cut vertices in $H'$, respectively. This statement, along with Property 1 of $H$, proves that $H'$ satisfies Property 3.

iii. Since $H$ satisfies Property 4 and $H'$ is a subgraph of $H$, $H'$ satisfies Property 4.

Hence, $H'$ is a PBCG of $G$.

**Lemma 4.** *Let $H'$ be a BCG of $G$ with mapping function $f1 : V(G) \to V(H')$ and $H''$ be a BCG of $H'$ with mapping function $f2 : V(H') \to V(H'')$. Then, $H''$ is a BCG of $G$ with mapping function $f3 : V(G) \to V(H'')$ where $f3 = f2 \circ f1$.*

*Proof.* Let $f1 : V(G) \to V(H')$ and $f2 : V(H') \to V(H'')$ are the mapping functions for $H$ and $H'$ respectively. We choose function $f3 : V(G) \to V(H'')$ which is $f2 \circ f1$ as the mapping between $G$ and $H''$. Let $u$, $v$, and $w$ be three distinct vertices in $G$. Let $u' = f1(u), v' = f1(v), w' = f1(w)$ and $u'' = f2(u'), v'' = f2(v'), w'' = f2(w')$.

1. Let $u$ be a cut vertex in $G$. By Property 1, $u$ is the only vertex in $G$ which maps to $u'$. We know $u'$ is a real cut vertex (also cut vertex) in $H'$ by Property 3. By Property 1, $u'$ is the only vertex in $H'$, which maps to $u''$ in $H''$. Hence $u$ is the only vertex which maps to $u''$, which is $f3(u)$. Therefore, Property 1 is satisfied.
2. Let $u$ and $v$ be two non cutvertices from different BCC in $G$. We know $f1(u) \neq f1(v)$ in $H$. At least one cut vertex exists, let $w$, which separates $u$ and $v$. $w'$ separates $u'$ and $v'$. So, $u'$ and $v'$ belong to two different BCC in $H$. $u'$ and $v'$ need not be noncut vertices. By Properties 1 and 2 of $f2$, we can conclude $f2(u') \neq f2(v')$. Hence $f3(u) \neq f3(v)$.
3. A cut vertex $w$ separates $u$ and $v$ in $G$ if and only if a real cut vertex $w'$ separates $u'$ and $v'$ in $H$. A cut vertex $w'$ separates $u'$ and $v'$ in $H'$ if and only if a real cut vertex $w''$ separates $u''$ and $v''$ in $H''$. Hence, a cut vertex $w$ separates $u$ and $v$ in $G$ if and only if a real cut vertex $w''$ separates $u''$ and $v''$ in $H''$.
4. We know for every edge $e(u'', v'')$ in $H''$ there exists at least one edge between any vertex from $f2^{-1}(u'')$ and any vertex form $f2^{-1}(v'')$ in $H'$. Let that edge be $(u', v')$. We also know for this edge, there exists at least one edge between any vertex from $f1^{-1}(u')$ and any vertex from $f1^{-1}(v')$ in $G$. For $e$ there exists at least one edge between any vertex from $f3^{-1}(u'')$ and any vertex form $f3^{-1}(v'')$ in $G$, and hence Property 4 is satisfied.

Hence $H''$ is a BCG of $G$.

**Lemma 5.** *Let $G$ be a graph. $G$ is a BCG of itself with a mapping function $f$, where for each $u \in V(G)$, $f(u) = u$.*

**Lemma 6.** *For $i \in \{2, \ldots, l\}$, let $H_i$ be a graph obtained after processing batch $B_i$ of edges on $H_{i-1}$ using Lines 4-8 of Algorithm 1. If $H_{i-1}$ is a BCG of $G_{i-1}$, then $H_i$ is a BCG of $G_i$.*

*Proof.* $G_{i-1}$ is connected. We recall $G_i = G_{i-1} + B_i$. Given $H_{i-1}$ is a BCG of $G_{i-1}$. Line 4 of Algorithm 1 adds transformed batch of edges to $H_{i-1}$, that means $H_{i-1} = H_{i-1} + B'$. Therefore, by Lemma 1, $H_{i-1}$ is a BCG of $G_i$ and $M : V(G_i) \to V(H_{i-1})$ is the mapping function. Line 5 of Algorithm 1 applies COMPRESS-GRAPH on $H_{i-1}$ and we get $H_i$ by transforming edges of $H_{i-1}$ with $I : V(H_{i-1}) \to V(H_i)$. Hence $H_i$ is a BCG of $H_{i-1}$ by Lemma 2. By Lemma 4, $H_i$ is a BCG of $G_i$ and mapping should be $I \circ M$ which is updated in Line 7. Line 9 applies TV-FILTER on $H_i$ and removes non-essential edges. By Lemma 3 and Lemma 4, $H_i$ is BCG of $G_i$ and TV-FILTER on $H_i$ gives BCG of $H_i$, so $H_i$ obtained at the end of iteration $i$ in the Algorithm 1 is a BCG of $G_i$.

**Theorem 1.** *Algorithm 1 identifies BCCs of $G$ correctly.*

*Proof.* Line 1 of the Algorithm 1 correctly constructs a spanning tree of $G$, which is represented by $H_0$. We recall that $G_0$ is the same as $H_0$ before the second pass begins. Line 2 initializes mapping $M$ as an identity function. By Lemma 5, $H_0$ is a BCG of $G_0$ with mapping $M$. After $k$ iterations in the algorithm are over, we are ready with a compressed graph $H_k$ and the updated mapping function $M$. By Lemma 6, $H_k$ is a BCG of $G$ with mapping function $M$. BCC-REPAIR subroutine correctly computes the label and component head arrays of graph $H_k$ that satisfy the following property: a subgraph induced on all vertices having the same label along with its component head forms a maximal subgraph of $H_k$ with no real cut vertices. By Property 3 of BCG, subgraph induced on inverse maps of vertices having the same label and inverse map of the component head in graph $G$ form a maximal subgraph with no cut vertices in $G$. Hence, vertices having the same label along with the component head of the label form BCCs in $G$.

## 5 Experimentation

In this section, we introduce the datasets and the configuration of the experimental platform. We then analyse the performance and scalability of the proposed algorithms compared to the baseline algorithms. Finally, we highlight key insights and the influencing factors of the GEM-BCC algorithm.

### 5.1 Experimental Setup

All experiments were conducted on a NVIDIA A100 GPU (40 GB memory, 2.04 TB/s memory bandwidth, 6 MB L2 cache, and 128 KB L1 cache per Streaming Multiprocessor, with 80 SMs total), paired with an AMD EPYC 7742 CPU (64 cores, 4 MB L1 cache, 32 MB L2 cache, and 256 MB L3 cache) supporting hyperthreading for up to 128 simultaneous threads. All programs were implemented in C++ and CUDA using 1024 threads per CUDA block, and CUDA streams were

Table 1: List of graph datasets along with their statistics used in our experiments.

| Datasets | \|V\| | \|E\| | \|BCC\| | Baselines | | | Ours | abv. |
|---|---|---|---|---|---|---|---|---|
| | | | | F-BCC$_{cpu}$ | WK | F-BCC$_{gpu}$ | E-BCC | |
| **Road Networks** | | | | | | | | |
| road_usa | 23M | 57.7M | 7.3M | 0.9 | 0.8 | 0.2 | 0.2 | RU |
| europe_osm | 50M | 108.1M | 43.4M | 1.2 | 0.94 | 0.4 | 0.4 | EO |
| **Web Crawls** | | | | | | | | |
| Webbase-2001 | 118M | 1.01B | 30.4M | 4.3 | OOM | OOM | 1.3 | WB |
| IT-2004 | 41M | 2.05B | 4.9M | 2.1 | OOM | OOM | 1.1 | IT |
| SK-2005 | 50M | 3.62B | 4M | 2.3 | OOM | OOM | 1.7 | SK |
| **Social Networks** | | | | | | | | |
| Twitter7 | 41M | 2.41B | 1.9M | 8.7 | OOM | OOM | 1.4 | TW |
| com-Friendster | 65M | 3.61B | 14M | 12.3 | OOM | OOM | 2.6 | CF |
| **Random Graph** | | | | | | | | |
| GAP-URAND | 134M | 4.29B | 1 | 34.2 | OOM | OOM | 2.6 | GU |
| **Biomedical Hypothesis Generation Systems** | | | | | | | | |
| Moliere-2016 | 30M | 6.68B | 1.4M | 16.8 | OOM | OOM | 2.7 | ML |
| Agatha-2015 | 183M | 11.6B | 18.4M | 40 | OOM | OOM | 2.5 | AG |

used to overlap data transfers with computation (i.e., transferring data for the next batch in the background while processing the current batch).

Although TV-FILTER theoretically bounds the number of edges to $O(n)$, its overhead can be significant in practice primarily due to BFS. To avoid this, we only apply TV-filter selectively. We pre-allocate space for up to $2 \times$ batch-size edges on the GPU, and only invoke TV-FILTER if the graph exceeds this capacity after multiple iterations without shrinking. To assess the performance and scalability of our algorithms, we selected ten diverse, publicly available datasets [4], [10], [12] [1]. These graphs range in size from around 50 million to 10 billion edges, as summarized in Table 1. Each experiment was run ten times and the reported results are the averages of these trials. The source code, data, and / or other artifacts have been made available at github.

### 5.2 Comparison with state-of-the-art BCC Algorithms

We evaluated our proposed algorithm, GEM-BCC (E-BCC), by comparing it against two publicly available baseline methods: FAST-BCC on CPU (F-BCC$_{cpu}$) and Wadwekar & Kothapalli's GPU-based algorithm (WK), along with a GPU variant of Fast-BCC (F-BCC$_{gpu}$).

We use the notation OOM (Out of Memory) in Table 1 to indicate instances where the WK algorithm was unable to execute due to insufficient GPU memory. The reason for this is that WK algorithm requires substantial memory not only to store the input graph but also to manage auxiliary data structures resulting in $O(n+m)$ space. A detailed analysis reveals that each vertex generally requires at least 8 bytes, and each edge needs a minimum of 16 bytes for representation. Additionally, WK allocates approximately 810 auxiliary arrays proportional to

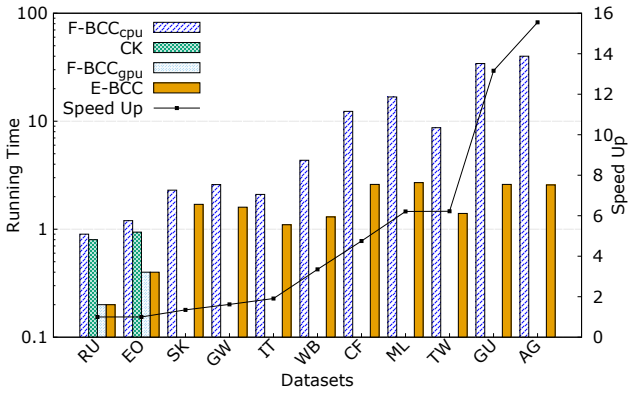


Fig. 3: Speedup comparison of `E-BCC`, `F-BCC`$_{cpu}$, `WK`, and `F-BCC`$_{gpu}$. Empty plots indicate instances where the respective algorithm failed to execute due to out-of-memory (OOM) errors.

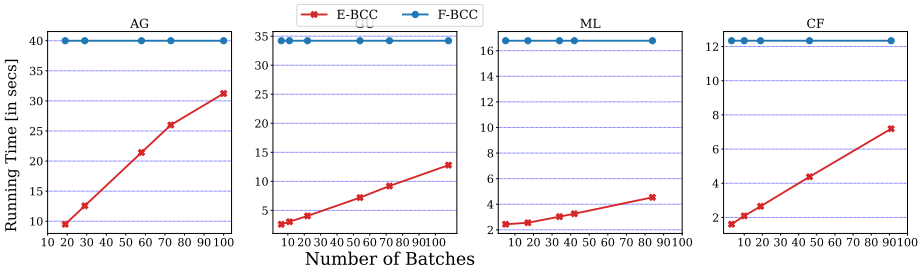


Fig. 4: Running Time vs Number of Batches

the vertex count and few to the edge count, significantly exacerbating memory usage. In contrast, due to the higher available CPU memory, the F-BCC algorithm successfully executed across all tested graph instances.

Notably, while the state-of-the-art GPU algorithm (WK) frequently exhausted available memory, our E-BCC algorithm consistently processed all tested graphs without issue. As shown in Figure 3, E-BCC demonstrates significant performance gains over the multicore F-BCC, achieving a maximum speedup of 15x and an average speedup of about 6x across most datasets. For the RU and EO datasets, however, the speedup remains close to 1; as for speed up, we compare E-BCCs runtime to that of the best-performing baseline. When the data fits into GPU memory, E-BCC effectively matches the best static GPU-based BCC algorithm, as it avoids many of the overheads otherwise involved.

### 5.3 Impact of different batch sizes

In this section, we examine how the algorithms running time varies with different amounts of GPU memory. As the GPU memory capacity increases, the number of batches decreases. Figure 4 illustrates that fewer batches lead to shorter running times, primarily because the algorithm is repeated over fewer batches. This trend aligns with our theoretical expectation as the depth of our algorithm depends on the number of batches. Consequently, in practice, it is most efficient to utilize the maximum available GPU memory to minimize the number of batches. Notably, we were able to process four of our largest datasets using as small as 10 GB of GPU memory.

## 6 Conclusion

We introduced GEM-BCC, a GPU based out-of-memory algorithm that processes large graphs exceeding the on-board memory by using batch processing. Although its runtime can be comparable to multicore CPU implementations in some scenarios, the primary goal is to enable GPU-based processing of large datasets that traditional in-memory algorithms cannot handle. We provided comprehensive proof of correctness and demonstrated the scalability for diverse set of datasets.

## References

1. Ahmed, N., Rossi, R.: The network data repository with interactive graph analytics and visualization. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015) (2015), https://networkrepository.com
2. Ben-David, N., Blelloch, G., Fineman, J., Gibbons, P., Gu, Y., McGuffey, C., Shun, J.: Implicit decomposition for write-efficient connectivity algorithms. In: International Parallel and Distributed Processing Symposium (IPDPS 2018). pp. 711–722. IEEE (2018). https://doi.org/10.1109/IPDPS.2018.00080
3. Cong, G., Bader, D.: An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps). In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005). pp. 45–54. IEEE (2005). https://doi.org/10.1109/IPDPS.2005.307
4. Davis, T., Hu, Y.: The university of florida sparse matrix collection. ACM Transactions on Mathematical Software **38**(1), 1–25 (2011). https://doi.org/10.1145/2049662.2049663
5. Dong, X., Wang, L., Gu, Y., Sun, Y.: Provably fast and space-efficient parallel biconnectivity. In: Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP 2023). pp. 52–65. ACM (2023). https://doi.org/10.1145/3572848.3577488
6. Hochbaum, D.: Why should biconnected components be identified first. Discrete Applied Mathematics **42**(2), 203–210 (1993). https://doi.org/10.1016/0166-218X(93)90046-Q

7. Hong, C., Dhulipala, L., Shun, J.: Exploring the design space of static and incremental graph connectivity algorithms on gpus. In: International Conference on Parallel Architectures and Compilation Techniques. pp. 55–69 (2020). https://doi.org/10.1145/3410463.3414658
8. Hopcroft, J., Tarjan, R.: Efficient algorithms for graph manipulation. Communications of the ACM **16**(6), 372–378 (1973). https://doi.org/10.1145/362248.362272
9. JáJá, J.: Parallel Algorithms. Addison-Wesley (1992)
10. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu (2014), last accessed 2023/10/25
11. Li, S., Tang, R., Zhu, J., Zhao, Z., Gong, X., Wang, W., Zhang, J., Yew, P.C.: Liberator: A data reuse framework for out-of-memory graph computing on gpus. IEEE Transactions on Parallel and Distributed Systems **34**(6), 1954–1967 (2023). https://doi.org/10.1109/TPDS.2023.3268662
12. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP 2013). pp. 456–471. ACM (2013). https://doi.org/10.1145/2517349.2522739
13. Sabet, A.H.N., Zhao, Z., Gupta, R.: Subway: Minimizing data transfer during out-of-gpu-memory graph processing. In: Proceedings of the 15th European Conference on Computer Systems (EuroSys 2020). pp. 12:1–12:16. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3342195.3387537
14. Sahu, S., Mhedhbi, A., Salihoglu, S., Lin, J., Özsu, M.T.: The ubiquity of large graphs and surprising challenges of graph processing: extended survey. The VLDB journal **29**, 595–618 (2020)
15. Sengupta, D., Song, S.L., Agarwal, K., Schwan, K.: Graphreduce: processing large-scale graphs on accelerator-based systems. In: International Conference for High Performance Computing, Networking, Storage and Analysis. SC '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2807591.2807655
16. Tang, R., Zhao, Z., Wang, K., Gong, X., Zhang, J., Wang, W., Yew, P.C.: Ascetic: Enhancing cross-iterations data efficiency in out-of-memory graph processing on gpus. In: International Conference on Parallel Processing (ICPP 2021). pp. 41:1–41:10. ACM (2021). https://doi.org/10.1145/3472456.3472457
17. Tarjan, R., Vishkin, U.: An efficient parallel algorithm for graph biconnectivity. SIAM Journal on Computing **14**(4), 862–874 (1985). https://doi.org/10.1137/0214061
18. Wad, C., Kothapalli, K.: A fast gpu algorithm for biconnectivity computation. In: Proceedings of the 10th International Conference on Contemporary Computing (IC3 2017). pp. 1–6. IEEE (2017). https://doi.org/10.1109/IC3.2017.8284293