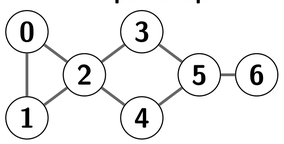
# A Multi-Phase GPU Pipeline for Biconnected Components

Soumendra Nath Banerjee, Turja Kundu, and Sanjukta Bhowmick
*University of North Texas*
SoumendraBanerjee@my.unt.edu, turjakundu@my.unt.edu, Sanjukta.Bhowmick@unt.edu  Sriram Srinivasan
*Bowie State University*
sriram882009@gmail.com

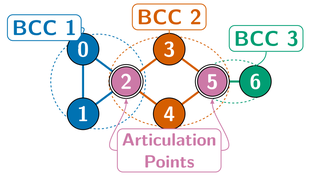
## Introduction

- Graph biconnectivity is a fundamental primitive for structural analysis, but scaling it to billion-edge graphs is hindered by inherently sequential algorithms like DFS.
- Existing parallel approaches often struggle with massive memory overheads or remain bottlenecked by tree-depth (graph diameter) during traversal.
- We propose a multi-phase GPU pipeline that reduces computational workload via aggressive 1-core peeling before executing a load-balanced, multi-root BFS.
- Our approach substantially outperforms CPU and earlier GPU baselines, and performs competitively against the state-of-the-art GPU approach, FastBCC-GPU.

**1. Input Graph**




**2. BCCs & APs**




## Related Work

- **Sequential DFS:** Hopcroft-Tarjan is optimal ($O(V+E)$) but is P-complete and inherently non-parallelizable.
- **Parallel/PRAM Methods:** Tarjan-Vishkin avoids DFS via an arbitrary spanning tree, but incurs $\Theta(m)$ memory overhead that exceeds GPU capacity on large graphs.
- **Recent GPU Baselines:** WK-17, WK-18, FAST-BCC, and FAST-BCC-Filter improve space efficiency using BFS trees or edge-pruning, but performance still depends heavily on graph diameter.
- **Our Advantage:** We combine 1-core filtering, which removes 29.5% of vertices on `road_usa` and averages 20–30% across all networks we test, with a depth-capped, multi-root approach that bounds both memory and sequential depth.

## Results at a Glance

| Baseline | Avg. speedup (ours) |
|---|---|
| WK-18 | 8.3× |
| WK-17 | 13.4× |
| FastBCC (CPU) | 10.1× |
| FastBCC-GPU | 0.86× (competitive) |

*Table. Average speedup across 17 real-world networks.*

## Method / Approach

- **Phase 1, 1-core peeling:** We remove degree-1 vertices and iteratively re-check affected neighbors until none remain, entirely on GPU.
- **Phase 2, multi-root capped BFS:** We seed random active vertices as tree roots and grow all trees in parallel. Atomic claiming guarantees each vertex joins exactly one tree.
- **Phase 3, per-tree BCC:** We resolve each tree's non-tree edges via an LCA walk, merging tree edges on the closed path into local BCCs.
- **Phase 4, cross-edge merge:** We resolve edges that span two trees to determine which local BCCs merge across tree boundaries and which stay separated by a true articulation point.

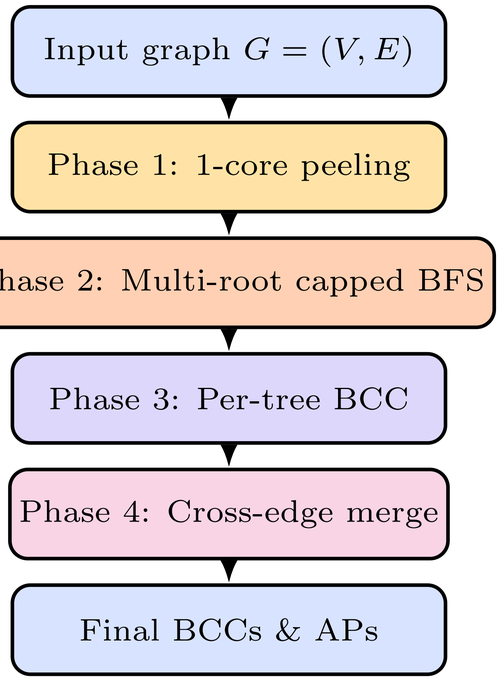
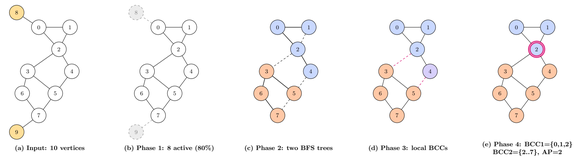



*Fig. Four-phase pipeline.*

## Mapping to the GPU

- **Peeling:** Threads scan vertices in parallel and decrement neighbor degrees atomically, so new degree-1 vertices appear without a host-side loop.
- **Multi-root BFS:** Many roots expose enough parallel work to fill the device. Atomic claiming resolves races between trees without locks.
- **Depth cap:** Capping tree depth bounds per-tree state and keeps memory below the $\Theta(m)$ overhead of Tarjan-Vishkin.
- **Phase 4:** Cross-edge resolution is our least parallel stage and the focus of our ongoing work.

## Example: Full Pipeline




- Phase 1 peels the two degree-1 pendants before any BCC computation begins.
- Cross edges 2–3 and 4–5 close a cycle spanning two trees, merging their local BCCs.

## Experimental Setup

- 17 real-world networks spanning web graphs, social networks, road networks, and citation/biological networks.
- Implementation: CUDA, GPU-resident throughout, no host-side per-vertex loops.
- Metrics: native stage-timer runtime (ms), peeled-vertex percentage, cross-tree edge count, and final BCC/articulation-point counts.

| Item | Configuration |
|---|---|
| GPU | NVIDIA A100, 40 GB |
| Host CPU | AMD EPYC 7763, 64 cores, 256 GB RAM |
| Software | CUDA 12.4, GCC 13.2.0 |
| Timing | Median of 3 runs after 1 warm-up; excludes graph loading |

## Results

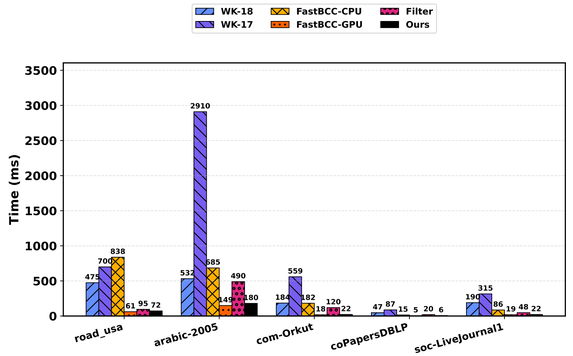



*Fig. Runtime (ms, log scale) on 5 representative networks. Lower is better.*

- We beat WK-18, WK-17, and CPU-based FAST-BCC on every network, often by an order of magnitude.
- Our 1-core peeling and load-balanced multi-root BFS drive this improvement by keeping every phase GPU-resident and avoiding sequential DFS entirely.
- Our performance when compared to FAST-BCC-GPU remains competitive.

## Ablation / Sensitivity

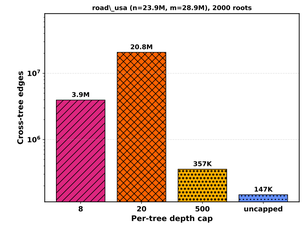



*Fig. Depth cap vs. cross-tree edges on road_usa.*

- A depth cap that is too small forces most vertices into single-vertex fallback trees, and cross-tree edges balloon to a large fraction of the whole graph.
- An adequately large cap nearly eliminates this fragmentation, but the right threshold depends on graph diameter and is not yet chosen automatically.
- This fragmentation directly drives Phase 4's cost, since Phase 4 must resolve every cross-tree edge.

## Limitations & Challenges

- We currently fix the per-tree depth cap and root count manually rather than deriving them from graph structure.
- Phase 4 (cross-edge merge) costs substantially more than Phases 1–3 combined on high-diameter graphs, and we identify it as our primary remaining bottleneck and the main source of our gap to FAST-BCC-GPU.
- Increasing root count did not meaningfully speed up Phase 2 in our testing, but it did increase fragmentation and, consequently, Phase 4's cost.

## Ongoing / Future Work

- **Close the gap:** We are replacing Phase 4's sequential dependency with a batched, GPU-native merge of local BCC labels, and we will report its speedup against FAST-BCC-GPU.
- **Auto-tuning:** We plan a lightweight preprocessing step that inspects graph structure and selects the depth cap, vertex cap, and root count per network.
- **Ablation:** We plan a systematic study across graph types to characterize how these three parameters interact.

## References

1. Hopcroft, Tarjan. *Efficient Algorithms for Graph Manipulation.* CACM, 1973.
2. Reif. *Depth-First Search is Inherently Sequential.* Info. Proc. Letters, 1985.
3. Tarjan, Vishkin. *An Efficient Parallel Biconnectivity Algorithm.* SIAM J. Comput., 1985.
4. Cong, Bader. *An Experimental Study of Parallel Biconnected Components Algorithms on SMPs.* IPDPS, 2005.
5. Wadwekar, Kothapalli. *A Fast GPU Algorithm for Biconnected Components.* IC3, 2017.
6. Kothapalli, Wadwekar. *Expediting Parallel Graph Connectivity Algorithms.* HiPC, 2018.
7. Dong, Wang, Gu, Sun. *Provably Fast and Space-Efficient Parallel Biconnectivity.* PPoPP, 2023.
8. Sahu, Aditya, Ramakrishna, Kothapalli, Banerjee. *GPU Algorithms for Biconnected Components on Large Graphs.* IPDPS, 2026.
9. Cheriyan, Thurimella. *Algorithms for Parallel k-Vertex Connectivity and Sparse Certificates.* STOC, 1991.