A Fast GPU Algorithm for Biconnected Components

by

Mihir Wadwekar, Kishore Kothapalli

in

2017 Tenth International Conference on Contemporary Computing (IC3-2017)

Report No: IIIT/TR/2017/-1

Centre for Security, Theory and Algorithms
International Institute of Information Technology
Hyderabad - 500 032, INDIA
August 2017

A Fast GPU Algorithm for Biconnected Components

Mihir Wadwekar, Kishore Kothapalli
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India 500 032.
Email: mihir.wadwekar@research.iiit.ac.in, kkishore@iiit.ac.in

***Abstract*— Finding the articulation points and the biconnected components of an undirected graph has been a problem of huge interest in graph theory. Over the years, several sequential and parallel algorithms have been presented for this problem. Our paper here presents and implements a fast parallel algorithm on GPU which is to the best of our knowledge the first such attempt and also the fastest implementation across architectures. The implementation is on an average 4x faster then the next best implementation. We also apply an edge-pruning technique which results in a further 2x speedup for dense graphs.**

Keywords - Articulation Points, Biconnected Components, Parallel, GPU

I. Introduction

A graph G is called biconnected if there exists at least two vertex disjoint paths between any two vertices of the graph. The maximal biconnected subgraphs of G are often called as the biconnected components of G. The problem of finding the biconnected components of a graph has long been a part of the algorithmic graph theory with applications to social networks [22], clustering [16], data visualization [18] and many other areas. An articulation point, which is a vertex whose removal disconnects the graph, represents a critical point in a network whose failure can disrupt the flow of messages across the network. In social networks, articulation points can indicate people who connect people of different interests. Biconnected graphs are fault-tolerant in the sense that the failure of a single node does not disable the network.

The classical solution to this problem is due to Tarjan which uses a depth first traversal of the underlying graph [21]. Given that performing a depth first traversal of a graph is a P-complete problem [9], the algorithm of Tarjan does not lend itself to efficient parallelization. The first parallel algorithm for this problem is designed by Tarjan and Vishkin [17] which converts the problem of finding the biconnected components of a graph to finding the connected components of an auxiliary graph. However, the approach of Tarjan and Vishkin suffers in practice due to the huge size of the auxiliary graph and also the operations required to construct the auxiliary graph. Cong and Bader [5] uses the connection between spanning trees and k-connecitivity of a graph [8] to improve the algorithm of Tarjan and Vishkin [17].

Given the overwhelming number of applications and significance of the problem, there is a renewed interest in this problem in the parallel setting. Algorithms for this problem have been studied on modern architectures such as multicore CPUs and the XMT [6], [11], [12], [7]. Recently, Slota and Madduri [6] proposed two parallel algorithms for finding the biconnected components of a graph. These algorithms are best suited for multi-core architectures and are since improved by the work of Chaitanya and Kothapalli [12]. The algorithm of [12] is particularly suited for sparse graphs.

GPUs have currently established themselves as an attractive and viable computing platform owing to several reasons. For instance, their massively parallel architecture consisting of thousands of smaller, more efficient cores can provide a significantly higher speedup. However due to the difference in architecture between a CPU and a GPU, algorithms designed for multi-cores CPUs are not always well-suited for GPUs. Such a situation has meant that CPU algorithms have to be sometimes reinterpreted to arrive at efficient algorithms in practice. While there are several recent works that use the biconnected components of a graph to improve on the performance of graph algorithms on the GPU [2], [19], to the best of our knowledge there is no GPU centric solution for finding the biconnected components of a graph.

In this paper, we present and implement the first GPU algorithm for biconnected components. Through several optimizations and a clever implementation, on a K40C GPU we are able to achieve a speedup of up to 70x over the approach of Slota and Madduri [6] and up to 9x over the approach of Chaitanya and Kothapalli [11], [12]. Our approach also particularly suits sparse graphs as is the case with other existing current algorithms [6], [11], [12]. To extend our approach to dense graphs, we borrow an edge pruning technique mentioned by Cong and Bader [5]. This further speeds up the algorithm for dense graphs achieving an implementation which works for all kinds of graphs.

A. Motivation

The approach of Slota and Madduri [6] performs multiple breadth-first searches(BFSs). After the first BFS generates the BFS tree, each thread picks a vertex and checks via another BFS whether a child node can reach a node above the current vertex. These BFSs terminate when a vertex at a higher level is found.

Scheduling BFSs on each thread is not feasible on a GPU due to the irregularity of work involved in exploring neighbors of vertices. The same irregularity also arises at a warp level. A single warp can be assigned a node with hundreds of neighbors to expand while some other warp may be assigned a vertex of low degree. This leads to work

imbalance where some threads are doing extra work while others are idle.

GPUs are structured as collection of SMs (Streaming Multiprocessors) each having their own cores, registers and caches. A BFS can be scheduled per SM of a GPU but that is still not an efficient way. In [4], Merrill et al. argue that the most efficient way for exploring vertices is when fine-grained scan-based exploration is supplemented with coarser cooperative thread array-based and warp-based exploration. An entire thread block is used for exploring a single large-degree vertex. Scheduling a BFS per SM although feasible would not be efficient.

Besides due to the irregularity of work involved in BFS, they are often the bottleneck in GPU graph algorithms. In our approach, we found that a single BFS would consume 20% to 50% of total time. Thus it is lot easier to work with simple tree traversals in parallel than doing multiple parallel BFSs.

Hence, we use BFS only once to generate the BFS tree and then work on the tree through simple tree traversals in parallel.

B. Organization

The rest of the paper is organized as follows. Section II presents and discusses our algorithm GPU-BiCC. Section III handles the implementation details of our algorithm on GPU. Section IV presents the experimental results and their analysis. Section V further extends the algorithm by presenting an improvement for dense graphs along with its experimental results. Section VI provides some concluding remarks.

II. Our Algorithm - GPU-BiCC

We build upon the following two lemmas used by Chaitanya and Kothapalli [11], [12] for their CPU parallel algorithm called LCA-BiCC. Let, $V_{lca}(G)$ be the set of the lowest common ancestors (LCAs) of each non-tree edge of G with respect to some BFS tree T.

Lemma 1: Let G be a 2-edge-connected graph and let T be a BFS tree of G. If v is not in $V_{lca}(G)$, then v cannot be an articulation point of G.

A bridge is an edge whose removal disconnects the graph. Naturally, end-points of bridges are also articulation points. However, not all articulation points are end-points of bridges. Chaitanya and Kothapalli [11], [12] construct an auxiliary graph so that all the articulation points in the original graph become the end-points of bridges in the auxiliary graph. Based on the same principle and lemmas, we also obtain an auxiliary graph, albeit in a more efficient and simpler way.

We split G into its 2-edge connected components. (A 2-edge connected component of a graph G is a maximal subset of edges such that every pair of vertices in the component have at least two edge disjoint paths between them.) Let u be the LCA of a non-tree edge pq. Let x and y be the base vertices in the cycle induced by pq. The base vertices for a cycle C induced by a nontree edge e, with LCA u, are the neighbors of u in the cycle C. We then introduce an alias vertex u', add the edge uu' and replace the edges ux and uy with $u'x$ and $u'y$ respectively. Only a single alias vertex is introduced for cycles sharing a common base vertex.

Finding the cycles that share a common base vertex is a non-trivial problem. From an algorithmic view point, it is essential that an efficient technique be designed for this purpose. We map this problem to the problem of finding the connected components of a graph. We construct a new graph H as follows. Every vertex that is a base vertex in G is a vertex in H. Two vertices in H share an edge if these two vertices are the base vertices for some cycle in G.

Figure 1 demonstrates an example for the construction of the auxiliary graph. This leads to the second lemma regarding newly introduced alias vertices.

Lemma 2: Let G be a 2-edge-connected graph, T a rooted BFS tree of G with root as r, and G' be the auxiliary graph of G constructed. The following then are true.

(i) Vertex r is an articulation point in G iff r is the LCA of more than one non-tree edge of G' according to a BFS in G' from r, and r is also the end point of some bridge in G_0.

(ii) For vertices u in G' with $u \neq r$, u is an articulation point of G iff there is a bridge uv in G' with $u \in G$ and $v \notin G$.

The proof of the two lemmas are shown in [12], [11].

A. Algorithm

Our GPU algorithm for biconnected components (BiCC) can be stated in four steps as mentioned in Algorithm 1.

Algorithm 1: GPU Algorithm for BiCC

Input: Graph G
Output: BCC ID for each edge
1 Generate BFS tree *T* from *G*
2 Obtain $V_{lca}(G)$ set and bridges from *T*
3 Reconstruct *G* into G_0 around $V_{lca}(G)$
4 Mark Articulation Points and BCCs in G_0

Step 1: BFS

In this step, we obtain a BFS tree T of the input graph G.

Step 2: LCA and Bridges

In this step, we traverse up from the end-points of every non-tree edge in parallel till the lowest common ancestor is found. This forms the $V_{lca}(G)$ set. Mark each edge encountered while traversing. Since we have traversed through every cycle of *G*, the unmarked edges are the bridges of *G*.

Step 3: Constructing the Auxiliary Graph

The auxiliary graph is constructed as mentioned earlier. A salient point of construction of the auxiliary graph is that the added edges never form a cycle. Every vertex can at most have one alias vertex. Our construction helps us in saving a BFS and LCA traversal of graph in next step, as compared to the approach of Chaitanya and Kothapalli [11], [12].

Step 4: Marking Articulation Points and BCCs

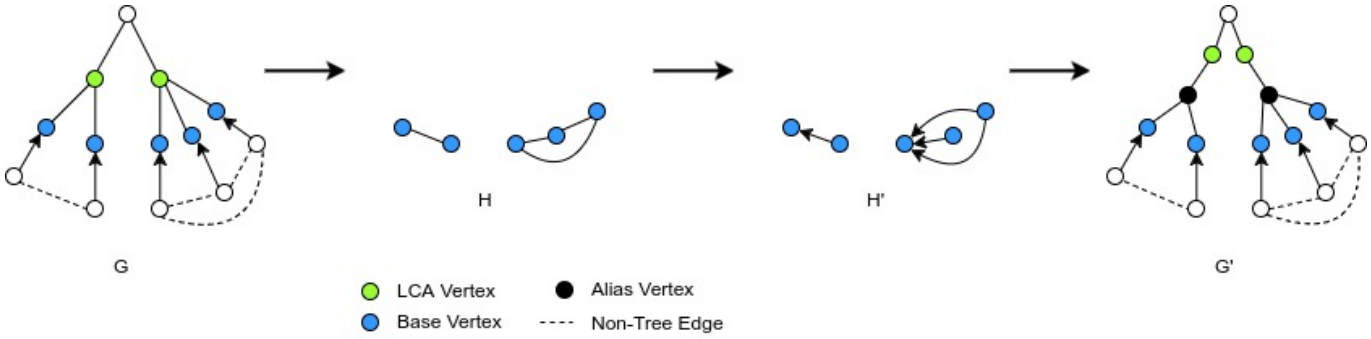

Fig. 1: H's vertex set is base vertices of G with edges induced by non-tree edges of G. H' is generated after applying connected components algorithm to H and contracting the trees. The unique ID for every connected component in H' serves as the ID for the alias vertex in G'.

Most of the work for this step can be done simultaneously in Steps 2 and 3. While traversing in Step 2, we record whether the traversal is finishing at that vertex or going further. If a traversal is finishing at a vertex, then it would finish even in the auxiliary graph. The alias vertices in our construction do not change the order or the number of vertices visited in traversal.

Now as Lemma 2 states, an LCA vertex is an articulation point if any of its incident edges with its alias vertex is a bridge. Let au be an edge between an LCA vertex a and its base vertex u. Let a' be the alias vertex introduced in the auxiliary graph. If au was a part of an unfinished tree traversal, then $ua'a$ would also be a part of that traversal. aa' cannot be a bridge. Thus all added edges can be checked for bridges by comparing with the information stored in Step 2. Figure 2 below illustrates the above example.

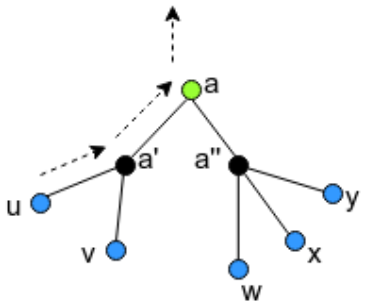

Fig. 2: Here a is an articulation vertex because aa^2 is a bridge. Vertex u had an upward traversal in Step 2 and hence aa^1 cannot be a bridge.

Furthermore in Step 2, each thread marks LCA vertex found for every non-tree edge. This value is accessible to every edge in the path to the LCA. To generate the unique BCC ID for each edge, each edge checks its LCA vertex. If the LCA vertex is an articulation point, then we have the ID, else we check for the LCA of the edge(LCA vertex, parent(LCA vertex)) and so on. Since we are traversing LCA of LCAs, the traversal is not long. The articulation vertex ID serves as the ID for the biconnected components.

Chaitanya and Kothapalli [11], [12] do another LCA traversal in Step 4 for getting the articulation points. As shown above, this traversal is unnecessary if some additional information is stored in Step 2. They also need to perform another BFS for getting the biconnected components. Through simple manipulation and better techniques used in the construction of the auxiliary graph, we are avoiding one BFS and one LCA traversal compared to the approach of [11], [12].

The implementation details are discussed in the next section.

B. Complexity Analysis

BFS or Step 1 takes $O(m+n)$ time sequentially. Step 2 involves tree traversals for each non-tree edge. Each tree traversal cannot exceed the diameter d of the graph as we are using a BFS tree. So for $m-n$ non-tree edges, Step 2 takes $O(d{\cdot}(m{-}n))$ time. For Step 3, we run connected components algorithm on the graph defined out of base vertices in G. In the worst case, the time taken by Step 3 is in $O(n+m)$. Step 4 involves each thread checking each alias vertex for bridges which would be $O(n)$. Thus, sequentially our algorithm takes $O(d \cdot (m-n))$ time. However, few real-world graphs have large diameter and if they do, very few LCA traversals consume $O(d)$ times. This is observed and mentioned by Chaitanya and Kothapalli [11], [12].

III. Implementation

On GPU, we schedule 1024 threads per block. The number of blocks then becomes $\frac{m-n}{1024}$. This configuration was found to be best over several trials. The rest of this section discusses specific GPU implementation details for each step of our algorithm.

Step 1: BFS

For the sake of modularity and ease, we adapt the BFS program from Merrill et al. [4] for our work. Merrill et al. [4] BFS works efficiently by employing block-based and warp-based exploration along with a fine-grained exploration. SMX-wide gathering is used for adjacency lists larger than warp width. Scan-based gathering collects the loose ends. They implement out-of-core vertex and edge frontiers, use local prefix sums instead of local atomic operations and use a best effort bit-mask for filtering.

As shown in their paper, their implementation achieves one of the fastest general implementation of BFS. Some other BFSs are suited for some particular instances, but we found Merrill et al. [4] work best for our general purposes. Our implementation however can benefit from any improvements in GPU based BFS in future

Step 2: LCA and Bridges.

Finding LCA's of non-tree edges are independent tasks and are hence easy to parallelize. Each thread picks a non-tree edge and traverses upwards till LCA is found. Although the memory accesses are not coalesced, each thread in its own is doing trivial work.

Each thread maintains three values while traversing. First, each thread marks every encountered edge. Thus, later bridges can be identified by gathering unmarked edges. Second, each thread marks whether it is ending at a vertex or going beyond it. This differentiation helps in identifying bridges in the auxiliary graph as explained in Step 4 of previous section. These two values can be marked and updated in the same array.

Third, each thread also stores its own ID in every edge it has discovered. In an separate array, every thread stores the LCA vertex it found at its ID location. Thus every edge can lookup its LCA vertex in a two-step read. Since multiple threads may traverse a tree edge, overwrites may occur. As long as the tree edges are pointing to some LCA, it does not matter. This LCA vertex later helps in marking BCCs. Figure 3 illustrates the two-step read for an example edge.

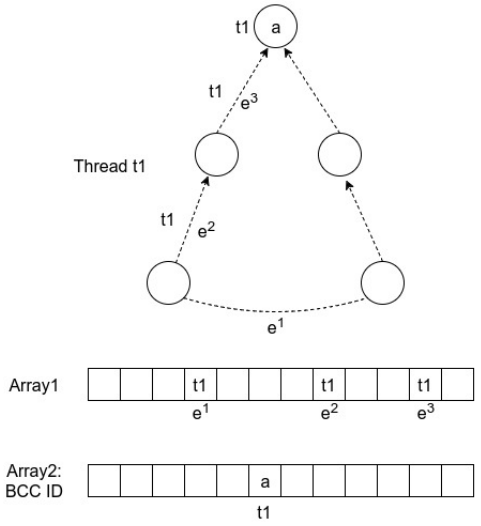

Fig. 3: Thread *t1* marks all edges it traversed with its ID. Then in Array2, it stores the LCA vertex found. Thus every edge of the graph knows its LCA vertex, by first looking up the thread which discovered it and then the corresponding value stored at that thread ID.

Step 3: Auxillary Graph Construction

As mentioned earlier, a connected components algorithm is required for identifying shared base vertices. We adapted Soman et al. [10] connected components work for our need. We found that their generalized GPU implementation had better timings than any other implementation. Although the code needed to be partially rewritten, the algorithm behind it remained the same as explained in their paper.

Once we know how the alias vertices are to be constructed, the actual construction is as follows. Each thread picks a non-tree edge and adds the corresponding edges for its alias vertex. Here also the work done by each thread in itself is simple.

Step 4: Articulation Points and BCCs

This step gets implemented mostly in Step 2, as threads are keeping a record of whether a traversal is stopping at an vertex or not. As mentioned in Step 4 of previous section, this information is enough to find the new bridges in the auxiliary graph. A simple kernel call with a lookup to stored global memory suffices. The articulation points get marked.

As for the BCCs, each edge has already got its own unique articulation point through Step 2. Any query regarding BCCs can be done in nearly linear time.

IV. Experiments And Analysis

A. Setup

We run our implementation on Nvidia Tesla K40C GPU. The K40C GPU features 2880 cores spread across 15 SMXs. It provides 12 GB memory with 64 KB of on-chip memory per SMX and achieves 1.4 Tflops of peak double precision floating point performance and a maximum throughput of 288 GB/sec. More details regarding Tesla K40C GPU can be found in [13]. We use CUDA 7.5 [14] for our implementation.

We run CPU-based algortihms on Intel Xeon E5-2650 CPU. This CPU is equipped with 128 GB RAM and a maximum memory bandwidth of 68 GB/s. The E5-2650 CPU features dual processors where each processor has 10 cores and each core can process two threads using hyper threading. Each core operates at 2.34 GHz which can be boosted to 3 GHz. The CPU offers 64 KB L1 cache, 256 KB L2 cache and a shared 25 MB L3 cache. These implementations were programmed using OpenMP [15]. We have compared our approach with Slota and Madduri [6], named BFS-BiCC in the plots, and also that of Chaitanya and Kothapalli [11], [12], named LCA-BiCC in the plots. These implementations were each run on 40 CPU threads.

The graphs used in our experiment were taken from the Stanford Large Network Dataset Collection(SNAP) [20] and the University of Florida Sparse Matrix Collection [23]. The graphs were mostly sparse in nature.

Graphs were assumed to be undirected and have a single connected component. Explicit edges if needed, were added to ensure connectivity in a preprocessing step. All experiments were repeated several times and average of the observations were used in plotting.

Results

BFS-BiCC does BFSs from every point in a BFS tree and checks whether a children node is accessible to node higher than the parent if the parent node is removed. If it is, then that node cannot be an articulation point. However Chaitanya and

Table 1: Graphs

Graph	Nodes	Edges	Diameter
webGoogle	875,713	5,105,039	21
webbase	1,000,005	3,105,536	29
amazon	262,111	1,234,877	32
webStandford	281,903	2,312,497	674
webBerkStan	685,230	7,600,595	514
roadNet-pa	1,088,092	1,541,898	786
roadNet-ca	1,965,206	2,766,607	849
netherlands-osm	2,216,688	4,882,476	2554
greatBritain	7,733,822	16,313,034	9340
asia-osm	11,950,757	25,423,206	48126

Kothapalli [11], [12] prove that only $V_{lca}(G)$ is potential set of articulation points. Hence BFS-BiCC can be modified to include only the LCA points. We implement this new version LCA-BFS-BiCC and tested it against our implementation. The below figure shows the runtime of the three algorithms LCA-BiCC, BFS-BiCC, LCA-BFS-BiCC against our GPU algorithm named GPU-BiCC in the plots.

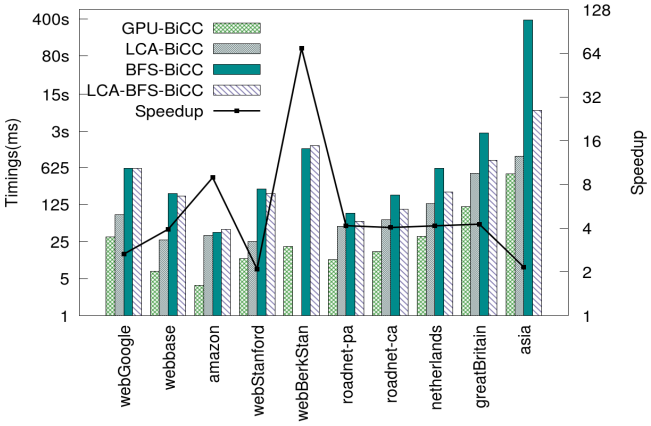

Fig. 4: The primary Y-axis represents time in milliseconds. The Secondary Y-axis gives the speedup of GPU-BiCC over the next fastest one.

GPU-BiCC performs on an average 4.03x faster than the next fastest implementation. In certain cases, it outperforms BFS-BiCC by nearly 700x and LCA-BiCC by 9x. LCA-BFS-BiCC appears to be faster than BFS-BiCC only in certain cases. In most of the other cases, LCA-BFS-BiCC performs similar if not worse than BFS-BiCC. This speedup in certain cases can be attributed to two factors First, in large sparse graphs, the number of non-tree edges are less and hence finding LCAs takes short time. However as graphs become dense, the LCA step becomes time-intensive. The advantage of working on a smaller subset of vertices is offset by actually finding the smaller subset of LCAs.

Second, it can be observed that even in cases of large sparse graphs, LCA-BFS-BiCC although better, still gives higher speedups in only certain cases. This speedup is influenced by the diameter of the graph.

BFS-BiCC slows quite considerably as the diameter increases. This can be verified by observing the diameter of a graph from Table 1 and its respective runtime from the above figure. As mentioned, BFS-BiCC does BFSs from each point to check whether a higher up point is reachable. In large diameter graphs, it results in a single thread traversing long chain of single-linked nodes causing work imbalance.

LCA-BFS-BiCC eliminates most of these vertices since only a small subset of nodes is considered. Experimentally we found that LCA subset of vertices is approximately 10% of the nodes. GPU-BiCC and LCA-BiCC have no such computation and are unaffected by the diameter of the graph. Thus it can be seen that despite the modifications to BFS-BiCC, GPU-BiCC performs better.

GPU-BiCC was also tested from various starting points. The BFS was run from maximum degree vertex among other random vertices. However no substantial change in timings was observed. GPU-BiCC performs consistently irrespective of the starting point.

V. Extension to Dense Graphs

GPU-BiCC performs one tree traversal from each non-tree edge of a BFS tree. In real-world graphs, which are usually sparse, the bottleneck step is the BFS. The number of non-tree edges are usually small and is in $O(n)$. However as the graph gets denser, the number of non-tree edges can get large. Since $O(m)$ GPU threads are launched for performing the tree traversals, this step slows down and becomes the most time consuming step.

Since BFS-BiCC performs BFSs from every vertex, as long as the average degree is less, BFS-BiCC remains unaffected by dense graphs. Dense graphs also generally have a low diameter and thus BFS-BiCC does not suffer from the large diameter drawback mentioned in above section. As a result the speedup observed by GPU-LCA-BiCC over BFS-BiCC is relatively less for dense graphs.

Cong and Bader [5] mention an edge-pruning technique while presenting their algorithm for finding the biconnected components of a graph. The key idea of the technique from [5] shows that most of the edges are non-essential for finding BCCs. Their pruning technique is as follows.

Consider graph G and its BFS tree T. Let G^1 be $G \setminus T$ and F be the spanning forest of G^1. Bader and Cong [5] then prove that the non-tree edges in G which are not in F are non-essential for biconnectivity.

Notice that a spanning forest of n nodes would have at most $n-1$ edges. Thus if the above technique is applied, then even for dense graphs, the number of LCA traversals to be performed drops down from $O(m)$ to $O(n)$.

We apply this pruning technique again using modified Soman et al. [10] connected components approach to generate a spanning forest and then tested it on dense graphs.

We used random graph generators for generating dense graphs. GTgraph suite [3] provides three random graph generators based on Erdös-Rényi [1] and the RMAT model. The graphs used are listed below.

The results of the pruning technique can be seen in Figure 5. The new algorithm is named as N-GPU-BiCC.

Table 4: Dense Graphs

Graph	Nodes	Edges	Diameter
liveJournal	4,847,571	68,993,773	9
com-Orkut	3,072,441	117,185,083	16
RMAT1M_50M	1,000,000	50,000,000	5
RMAT1M_70M	1,000,000	70,000,000	8
R500K_50M	500,000	50,000,000	5
R1M_25M	1,000,000	25,000,000	13
D1M_50M	1,000,000	50,000,000	5008
D2M_75M	2,000,000	75,000,000	11053

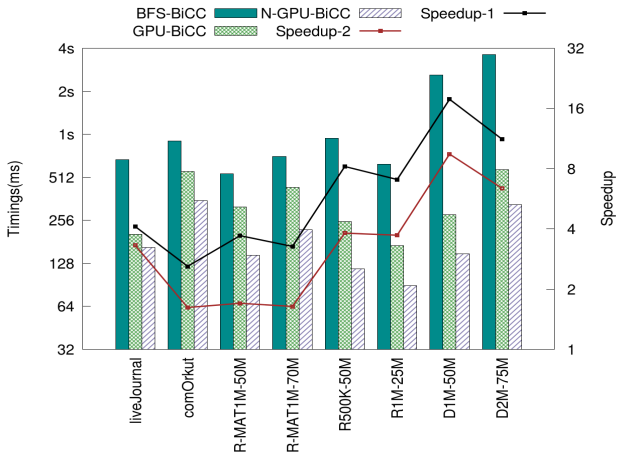

Fig. 5: Primary Y-axis represents the timings in milliseconds. On Secondary Y-axis, Speedup-1 represents speedup of N-GPU-BiCC over BFS-BiCC while Speedup-2 shows speedup of GPU-BiCC over BFS-BiCC.

On an average, N-GPU-BiCC achieves a 2x speedup over GPU-BiCC for dense graphs. N-GPU-BiCC prunes *m-n* edges. For sparse graphs, *m-n* is not a big number and hence a significant speedup is not observed. The cost of pruning the edges slows down N-GPU-BiCC for sparse graphs. Experimentally, we found that the overhead for pruning the edges is worth it when *m* roughly crosses 10 times *n*.

VI. Conclusion

In this paper we have presented a fast GPU algorithm for finding biconnected components in a graph. Our algorithm avoids an additional BFS and a LCA traversal step as compared to Chaitanya and Kothapalli [11], [12]. Our construction of auxiliary graph is also more efficient. Our implementation is on an average 4x faster than the next best implementation.

We have further applied Cong and Bader's edge pruning technique [5] which has resulted in a further 2x speedup for dense graphs.

Our implementation(to the best of our knowledge) is fastest for bi-connected components for all types of graphs.

References

[1] B. Bollobas. Random Graphs. Cambridge University Press, 2001.

[2] C. Pachorkar, M. Chaitanya, K. Kothapalli and D. Bera. Efficient Parallel Ear Decomposition of Graphs with Application to Betweenness-Centrality. In Proc. of IEEE HiPC, 2016.

[3] D. Bader and K. Madduri. GTgraph: A suite of synthetic graph generators

[4] D. G. Merrill, M. Garland and A. S. Grimshaw. Scalable GPU Graph traversal. PPoPP' 12, 117-128, 2012.

[5] G. Cong and D. Bader. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors(smps). IEEE IPDPS, 2005.

[6] G. Slota and K. Madduri. Simple parallel biconnectivity algorithms for multicore platforms. HiPC, 2014.

[7] J. A. Edwards and U. Vishkin. Better Speedups Using Simpler Parallel Programming for Graph Connectivity and Biconnectivity, in Proc. of ACM PMAM, 2012.

[8] J. Cheriyan and R. Thurimella. Algorithms for Parallel k-Vertex Connectivity and Sparse Certificates, in Proc. of ACM SPAA, pp. 391–401, 1991.

[9] J. JaJa. An Introduction To Parallel Algorithms, Addison-Wesley, 2004.

[10] J. Soman, K. Kothapalli and P. J. Narayanan. A Fast GPU Algorithm for Graph Connectivity. IPDPSW, 2010.

[11] M. Chaitanya and K. Kothapalli, A Simple Parallel Algorithm for Biconnected Components in Sparse Graphs. IPDPS Workshops, 2015.

[12] M. Chaitanya and K. Kothapalli, Efficient Multicore Algorithms For Identifying Biconnected Components, in International Journal of Networking and Computing, 2016.

[13] Nvidia Tesla K40C http://www.nvidia.in/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-00_v03.pdf.

[14] Nvidia CUDA http://docs.nvidia.com/cuda/

[15] OpenMP http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[16] R. Butafogo and B. Schneiderman. Identifying aggregates in hypertext structures, in: Proc. 3rd ACM Conference on Hypertext, 1991.

[17] R. Tarjan and U. Vishkin. An Efficient Parallel Biconnectivity Algorithm. SIAM J. Comput., 14(4), 862874, 1985.

[18] SAS(R) OPTGRAPH http://support.sas.com/documentation/cdl/en /procgralg/68145/PDF/default/procgralg.pdf

[19] S. Nayyorodden, M. Gambhir, and K. Kothapalli. A Study of Graph Decomposition Algorithms for Parallel Symmetry Breaking, in Proc. of IEEE IPDPS Workshops, 2017.

[20] Stanford Network Analysis Platform Dataset. http://www.cise.u.edu/research/sparse/matrices/SNAP.

[21] T. Cormen, C. Leiserson, R. Rivest and C. Stein. Introduction To Algorithms, MIT Press, 2001.

[22] U. Brandes, D. Wagner. visone - Analysis and Visualization of Social Networks. In M Junger, P Mutzel (eds.), Graph Drawing Software, pp. 321–340. Springer-Verlag, 2003.

[23] Univeristy of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.