TurboBC: A Memory Efficient and Scalable GPU Based Betweenness Centrality Algorithm in the Language of Linear Algebra

Oswaldo Artiles
oarti001@fiu.edu
School of Computing and Information Sciences, Florida International University
Miami, Florida, USA

Fahad Saeed
fsaeed@fiu.edu
School of Computing and Information Sciences, Florida International University
Miami, Florida, USA

ABSTRACT

Betweenness centrality (BC) is a shortest path centrality metric used to measure the influence of individual vertices or edges on huge graphs that are used for modeling and analysis of human brain, omics data, or social networks. The application of the BC algorithm to modern graphs must deal with the size of the graphs, as well with highly irregular data-access patterns. These challenges are particularly important when the BC algorithm is implemented on Graphics Processing Units (GPU), due to the limited global memory of these processors, as well as the decrease in performance due to the load unbalance resulting from processing irregular data structures. In this paper, we present the first GPU based linear-algebraic formulation and implementation of BC, called TurboBC, a set of memory efficient BC algorithms that exhibits good performance and high scalability on unweighted, undirected or directed sparse graphs of arbitrary structure. Our experiments demonstrate that our TurboBC algorithms obtain more than 18 GTEPs and an average speedup of 31.9x over the sequential version of the BC algorithm, and are on average 1.7x and 2.2x faster than the state-of-the-art algorithms implemented on the high performance, GPU-based, gunrock, and CPU-based, ligra libraries, respectively. These experiments also show that by minimizing their memory footprint, the TurboBC algorithms are able to compute the BC of relatively big graphs, for which the gunrock algorithms ran out of memory.

CCS CONCEPTS

• **Theory of computation → Massively parallel algorithms**; • **Mathematics of computing → Graph algorithms**.

KEYWORDS

GPU, CUDA, graph parallel algorithms, centrality, linear algebra.

ACM Reference Format:
Oswaldo Artiles and Fahad Saeed. 2021. TurboBC: A Memory Efficient and Scalable GPU Based Betweenness Centrality Algorithm in the Language of Linear Algebra. In *50th International Conference on Parallel Processing Workshop (ICPP Workshops '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3458744.3474047

1 INTRODUCTION

Centrality is a fundamental concept in graph analytics [2], used to measure the influence of individual vertices or edges on huge graphs that are used for modeling and analysis of human brain [17], omics data [4], or social networks [15]. One of the most important measures of centrality is the shortest-path based *betweenness centrality (BC),* a metric used to measure the importance of vertices and/or edges in a graph [8].

BC algorithms have enough parallelism to be implemented using all the computational power of modern Graphics Processing Units (GPU's) [7], however this implementation is challenging because real world graphs have some vertices whose degree are much greater than the mean degree in the graph, resulting in data-access patterns which are highly irregular. These type of data produces load imbalances and warp divergences that negatively affect the performance of the kernels in GPUs. The limited global memory and the data-transfer bottleneck of the GPU are also important challenges to implement scalable BC algorithms for the BC computation on modern huge graphs. These challenges result in limits in the scalability and performance of the BC algorithms, being therefore an active area of research [14, 16, 18–21].

In this paper, we propose TurboBC, a set of GPU-based BC algorithms in the language of linear algebra. The memory efficient and highly scalable BC algorithms on TurboBC are based on two parallel optimizations. Our first optimization was to reduce the space-complexity of the algorithm by limiting the number and the size of the arrays used on the computations performed by the GPU kernels. The second optimization was to design and implement our BC algorithms by exploiting the sparsity of the frontier and output vectors of the Breadth First Search (BFS) stage.

The main contributions of the paper are:

(1) We designed and implemented TurboBC, the first implementation of memory efficient and highly scalable GPU-based BC algorithms in the language of linear algebra. The TurboBC algorithms are applicable to unweighted, directed and undirected graphs represented by sparse adjacency matrices in the Compressed Sparse Column (CSC) and the transpose of the Coordinate Sparse (COO) formats. In order to reduce the memory footprint and to increase the memory efficiency and the scalability of TurboBC, the algorithms were designed

to use only one sparse storage format for each BC computation, also the number of auxiliary arrays on the device side was minimized. The reduction in the memory-footprint increased the memory bandwidth utilization and reduced the number of unnecessary floating operations. The design and implementation of the TurboBC algorithms also exploited the sparsity of the frontier and output vectors of the Breadth First Search (BFS) stage. These optimizations improved the performance and the scalability of the TurboBC algorithms.

(2) A comprehensive experimental detail and results are presented to assess the performance of the GPU-based BC algorithms in TurboBC. Our TurboBC algorithms obtained more than 18 GTEPs (billions of transverse edges per second), and an average speedup of 31.9x over the sequential version of the BC algorithm, and were on average 1.7x and 2.2x faster than the state-of-the-art algorithms implemented on the high performance, GPU-based, gunrock [21], and CPU-based, ligra [20] libraries, respectively. These experiments also showed that by minimizing their memory footprint, the GPU memory usage of of the gunrock library was higher than the memory usage of the TurboBC algorithms, allowing these algorithms to compute the BC of relatively big graphs, for which the gunrock algorithms ran out of memory. Our experiments also demonstrated that the performance obtained by the TurboBC algorithms, measured as MTEPs, as function of the GPU memory bandwidth, were much greater than those obtained by the BC algorithms in the gunrock library, showing that the GPU memory is used more efficiently by the TurboBC algorithms.

The remaining of this paper is organized as follows: Section 2 presents a general description of the BC algorithm. Section 3 presents details of the design and implementation of the TurboBC algorithms. Section 4 is dedicated to the experimental results. Section 5 describes the related work, and the summary and future work are presented in Section 6.

2 BETWEENNESS CENTRALITY ALGORITHM

The shortest-path betweenness centrality (BC) algorithm is applicable to any unweighted, directed or undirected graph $G = (V, E)$, where V is the finite set of vertices and E the set of edges. Any pair $(u, v) \in E$ implies that the vertices u and v in V are connected by an edge in G. A graph G is directed if E consists of ordered pairs, otherwise, G is undirected. Given a source vertex $s \in V$ in a graph G, the Breadth First Search (BFS) stage of the BC algorithm performs a systematic search of every vertex on E that is reachable from s. The algorithm computes the shortest path, i.e., the smallest number of edges from s to each reachable vertex t. The number of shortest paths between the vertices s and t is denoted by σ_{st}, and $\sigma_{st}(v)$ is equal to the number of shortest paths between s and t passing through the vertex $v \in V$, where v is different than s and t [1, 5].

Betweenness centrality of a vertex v, $BC(v)$, in a graph G was formally defined by Freeman [8] as

$$BC(v) = \sum_{s \neq v \neq t} \sigma_{st}(v)/\sigma_{st} = \sum_{s \neq v \neq t} \delta_{st}(v) \tag{1}$$

where $\sigma_{st}(v)/\sigma_{st} = 0$, if $\sigma_{st} = 0$, and $\delta_{st}(v) = \sigma_{st}(v)/\sigma_{st}$, the *pair-wise dependences*, is the fraction of shortest paths between the vertices s and t that pass through v. This definition of BC equally applies to disconnected and connected, directed and undirected graphs [8]. The straightforward computation of the BC of a vertex v, starts by computing the number and the length of all-pairs shortest paths over the graph, followed by computing the BC for each vertex by looking at all other pairs of vertices, and increasing the value of $BC(v)$ if the vertex, v, was in the corresponding shortest path. If $|V| = n$, the time complexity of this BC algorithm is $O(n^3)$, and its space complexity is $O(n^2)$.

Brandes [5], proposed a more efficient BC algorithm on which the pair-wise dependences can be aggregated without computing all of them explicitly. Let the one-sided dependences be defined as

$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v) \tag{2}$$

for all $s, v \in V$. Then

$$BC(v) = \sum_{s \neq v} \delta_s(v) \tag{3}$$

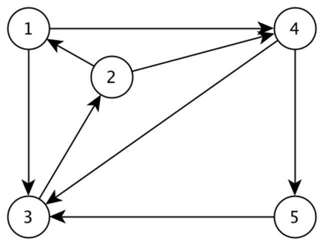
The following recurrence relation computes the one-sided dependences in the Brandes' BC algorithm

$$\delta_s(v) = \sum_{w:d(s,w)=d(s,v)+1} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)) \tag{4}$$

where $d(s, v)$ is the length of the shortest path from s to v. The recurrence relation 4 computes the one-sided dependence of a vertex s on some vertex v from the one-sided dependence of a vertex w one edge far away. For a graph with $|E| = m$, the time complexity of the Brandes' algorithm for unweighted graphs is: $O(nm)$, and the space complexity: $O(n + m)$. This algorithm is especially suitable for graphs represented by sparse adjacency matrices.

3 BC ALGORITHMS IN THE LANGUAGE OF LINEAR ALGEBRA FOR UNWEIGHTED GRAPHS.

This section describes the design and implementation of our GPU-based TurboBC algorithms in the language of linear algebra for unweighted graphs. The TurboBC algorithms were implemented for graphs represented by sparse adjacency matrices in the Compressed Sparse Column (CSC) format, as well in the COOC format which is the transpose of the Coordinate Sparse (COO) format. Both sparse formats are suitable to implement the sparse matrix-vector multiplication operations included in the BC Algorithm 1. Figure 1 shows an example of the CSC and COOC formats for a sparse adjacency matrix representing an undirected, unweighted graph. For a $n \times n$ adjacency sparse matrix A with m non-zero elements representing unweighted graphs, the array row_A (size m) of the CSC format, stores the corresponding row indices of the subsequent non-zero values of the columns in the matrix, and the array CP_A (size $n + 1$) stores the indices of the elements in the array row_A, that start a column. The first element of CP_A is always equal to 1 (one-based format) and the last element equal to $m + 1$. The COOC format contains two arrays: row_A which is equal to the corresponding array in the CSC format, and the col_A (size m) array that stores the column indices of the non-zero values of the adjacency matrix A. In order to reduce the memory footprint and increase the performance of the TurboBC algorithms, the arrays that stores the non-zero values of the binary sparse adjacency matrix of unweighted graphs were not used in the corresponding sparse matrix-vector multiplication (SpMV) operations of our algorithms.



$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

CSC

$$\mathbf{row_A} = \begin{bmatrix} 2 & 3 & 1 & 4 & 5 & 1 & 2 & 4 \end{bmatrix}$$
$$\mathbf{CP_A} = \begin{bmatrix} 1 & 2 & 3 & 6 & 8 & 9 \end{bmatrix}$$

COOC

$$\mathbf{row_A} = \begin{bmatrix} 2 & 3 & 1 & 4 & 5 & 1 & 2 & 4 \end{bmatrix}$$
$$\mathbf{col_A} = \begin{bmatrix} 1 & 2 & 3 & 3 & 3 & 4 & 4 & 5 \end{bmatrix}$$

Figure 1: Example of CSC and COOC sparse storage formats for a sparse adjacency matrix representing a directed, unweighted graph.

3.1 Regular and irregular graphs

We implemented two types of BC algorithms. The first type, called scalar algorithms, computes the sparse matrix-vector multiplication with GPU kernels which assign one thread per vertex (CSC format) or one thread per edge (COOC format). The second type of algorithms, called vector algorithms, computes the sparse matrix-vector multiplication with GPU kernels which assign one warp per vertex (CSC format). In this paper, we classified the graphs in two classes: *regular* graphs and *irregular* graphs. The regular graphs are those for which, in our experiments, the scalar BC algorithms obtained the best performance, while irregular graphs are those for which the vector BC algorithms obtained the best performance. We also used the scale free metrics, proposed in reference [13], to approximately quantify when a graph is regular or irregular. The scale free metrics scf, for a graph $G = (V, E)$ is defined by

$$scf = \sum_{(u,v)\in E} \text{degree(u)} * \text{degree(v)} \qquad (5)$$

where degree(u) is the degree of vertex $u \in V$, for directed graphs degree(u) = out.degree(u). Our experiments showed that for regular graphs the scf metric is in the range $[1, 224]$, and for irregular graphs in the range $[5846, 651837]$, more details about these results are given in Section 4 on which the experimental results are presented.

Algorithm 1 Linear algebra shortest path vertex betweenness centrality algorithm for a graph represented by a sparse adjacency matrix A in the COOC sparse storage format.

```
 1: Input: $\boldsymbol{A}$.                       ▷ sparse adjacency matrix representing a graph.
 2: Output: $\boldsymbol{\sigma}(1 \ldots n)$       ▷ stores number of shortest paths.
 3: Output: $\boldsymbol{bc}(1 \ldots n)$           ▷ betweenness centrality vector
 4: procedure BC-LA($G = \boldsymbol{A} : \mathbb{B}^{n\times n}$)
 5:     $\boldsymbol{bc} \leftarrow 0$
 6:     for $s \leftarrow 1, n$ do                 ▷ $s$: source vertex of BFS tree
 7:         $d \leftarrow 0$                        ▷ $d$: the current depth being examined
 8:         $c \leftarrow 1$                        ▷ c: check if the vector $\boldsymbol{f}$ is equal to 0
 9:         $S \leftarrow 0$                        ▷ stores depth at which a vertex is discovered
10:         $\boldsymbol{\sigma} \leftarrow 0$
11:         while $c > 0$ do                       ▷ BFS stage starts
12:             $d \leftarrow d + 1$
13:             $c \leftarrow 0$
14:             $\boldsymbol{f_t} \leftarrow 0$
15:             if $d == 1$ then
16:                 $f(s) \leftarrow 1$
17:                 $\sigma(s) \leftarrow 1$
18:             end if
19:             $\boldsymbol{f_t} \leftarrow \boldsymbol{A^T f}$
20:             if $\exists \sigma(i) == 0$ then
21:                 $f(i) \leftarrow f_t(i)$
22:             end if
23:             if $\exists f(i)! = 0$ then
24:                 $S(i) \leftarrow d$
25:                 $\sigma(i) \leftarrow \sigma(i) + f(i)$
26:                 $c \leftarrow 1$
27:             end if
28:         end while
29:         $d \leftarrow d - 1$
30:         $\boldsymbol{\delta} \leftarrow 0$
31:         while $d > 1$ do                       ▷ one-sided dependences vector stage starts
32:             $\boldsymbol{\delta_u} \leftarrow 0$
33:             $\boldsymbol{\delta_{ut}} \leftarrow 0$
34:             if $S(i) == d$ and $\sigma(i) > 0$ then
35:                 $\delta_u(i) \leftarrow (1.0 + \delta(i)) \div \sigma(i)$
36:             end if
37:             $\boldsymbol{\delta_{ut}} \leftarrow \boldsymbol{A^T \delta_u}$
38:             if $S(i) == d - 1$ then
39:                 $\delta(i) \leftarrow \delta(i) + \delta_{ut}(i) \times \sigma(i)$
40:             end if
41:             $d \leftarrow d - 1$
42:         end while
43:         for $v \leftarrow 1, n$ do                 ▷ update of vector bc starts
44:             if $v \neq s$ then
45:                 $bc(v) \leftarrow bc(v) + \delta(v)$
46:             end if
47:         end for
48:     end for
49:     return $\boldsymbol{bc}$
50: end procedure
```

3.2 BC algorithms

Algorithm 1 represents the linear algebra formulation of the Brandes' BC algorithm for a graph $G = (V, E)$ with n vertices and m edges, represented by $n \times n$ sparse adjacency matrix A in the COOC format, with m non-zero elements. This algorithm is inspired by the BC algorithm described on chapter 6 of reference [10]. Algorithm 1 computes the betweenness centrality vector, $\boldsymbol{bc}$, for all the connected vertices of the graph G using a two-stages procedure.

The first stage is a *forward* stage on which a Breadth First Search (BFS) from the source vertex s is performed at the first while loop (lines 11 to 28), where d represents the current depth of the discovered vertices. The final value of d is equal to the height of the BFS tree rooted at s. The output vector $\boldsymbol{\sigma}$ contains the number of shortest paths from the source vertex to the discovered vertices. The frontier vector $\boldsymbol{f}$ contains the number of shortest paths from the discovered vertices in the last iteration, to the undiscovered vertices to which there is some edge. The while loop stops when the vector $\boldsymbol{f}$ is equal to 0, i.e., when all the vertices reachable from s have been discovered. The vector $\boldsymbol{f}$ is updated by the sparse matrix-vector multiplication (SpMV) operation with the adjacency matrix (line 19), followed by a mask operation (lines 20 to 22) that exploits the sparsity of the vector $\boldsymbol{\sigma}$ and updates the shortest paths to vertices on $\boldsymbol{f}$, not yet contained on the vector $\boldsymbol{\sigma}$, guaranteeing that only the new discovered shortest paths are added to $\boldsymbol{\sigma}$ (line 25). By using the sparsity of the vector $\boldsymbol{f}$, the vectors $\boldsymbol{S}$ and $\boldsymbol{\sigma}$ are updated only when the corresponding component of the vector $\boldsymbol{f}$ is not zero (lines 23 to 27). The vector $\boldsymbol{S}$ stores the depth at which each vertex is discovered.

The second stage of Algorithm 1 is a *backward* stage on which the one-sided dependences vector, $\boldsymbol{\delta}$, is computed within the second while loop (lines 31 to 42), using Equation 4. For the computations in this stage, the vertices are visited in reverse order of their depth. The computation of the vector $\boldsymbol{\delta}$ starts when the auxiliary vector $\boldsymbol{\delta_u}$ is computed (lines 32 to 36) for those values derived from the children at depth d, which are stored on the vector S. The vector $\boldsymbol{\delta_u}$ is then weighted by the adjacency matrix A with the SpMV operation (line 37). The vector $\boldsymbol{\delta}$ is updated (lines 38 to 40), with the values corresponding at depth $d-1$ as determined by the vector $\boldsymbol{S}$. Finally, the betweenness centrality vector $\boldsymbol{bc}$ is computed, using Equation 3, for all parent vertices, v, not equal to the source vertex s (lines 43 to 47). For undirected graphs the computation of the vector $\boldsymbol{bc}$ should compensate by the double counting of every pair of vertices, hence $bc(v) \leftarrow bc(v) + \delta(v)/2$ for these graphs [5].

The BC algorithm with the sparse adjacency matrix in the CSC format has the same two stages of Algorithm 1, with the difference that in the first stage, the mask operation is included in the SpMV operation as shown in Algorithm 3.

Algorithm 2 Algorithm to implement the sequential SpMV operations of Algorithm 1 (lines 19 and 37) with the sparse adjacency matrix in the COOC format.

```
1: Input: $\boldsymbol{x}$,$\boldsymbol{row_A}$,$\boldsymbol{col_A}$
2: Output: $\boldsymbol{y}$
3: procedure scCOOC-SpMV($\boldsymbol{x}$,$\boldsymbol{row_A}$,$\boldsymbol{col_A}$,$\boldsymbol{y}$)
4:     for $k \rightarrow 1, m$ do
5:         if $x(row_A(k)) > 0$ then
6:             $y(col_A(k)) \leftarrow y(col_A(k)) + x(row_A(k))$
7:         end if
8:     end for
9: end procedure
```

3.3 Sparse matrix-vector multiplication (SpMV).

Our experimental results showed that the runtime of the SpMV operation (lines 19 and 37) can be up to 90 % of the total runtime of Algorithm 1, determining therefore the overall performance of the BC algorithm. We implemented the SpMV operation with three algorithms, the first one based in the COOC format and the other two based on the CSC format.

There are graphs with some vertices with a much higher degree than the mean value of the degrees in the graph, the SpMV operation for these graphs creates load unbalance in the threads of the GPU which negatively affected the performance of the SpMV algorithm. Our experiments showed that the SpMV algorithm based on the COOC format are less affected by this load unbalance, when applied to regular graphs which have vertices with much higher degrees than the mean degree of the graph. Algorithm 2 implements the sequential version of the SpMV operations on the first and second stages (lines 19 and 37) of Algorithm 1 with the sparse adjacency matrix in the COOC format. The sparsity of vector $\boldsymbol{x}$ is exploited by updating the vector $\boldsymbol{y}$ only when the corresponding component of vector $\boldsymbol{x}$ is greater than zero (line 5). The parallelization of Algorithm 2, known as COOC-scalar (scCOOC), on a GPU kernel assigns one thread per edge. In this paper, the acronym TurboBC-scCOOC designated the BC algorithm using the scCOOC algorithm for the SpMV operation.

Algorithm 3 Algorithm to implement the sequential SpMV operations of Algorithm 1 (lines 19 and 37) with the sparse adjacency matrix in the CSC format.

```
1: Input: $\boldsymbol{x}$,$\boldsymbol{CP_A}$,$\boldsymbol{row_A}$
2: Output: $\boldsymbol{y}$
3: procedure scCSC-SpMV($\boldsymbol{x}$,$\boldsymbol{CP_A}$,$\boldsymbol{row_A}$,$\boldsymbol{y}$)
4:     for $i \rightarrow 1, n$ do
5:         if $\sigma(i) == 0$ then
6:             $sum \leftarrow 0$
7:             $start \leftarrow CP_A(i)$
8:             $end \leftarrow CP_A(i+1) - 1$
9:             for $k \rightarrow start, end$ do
10:                $sum \leftarrow sum + x(row_A(k))$
11:            end for
12:            if $sum > 0$ then
13:                $y(i) \leftarrow sum$
14:            end if
15:        end if
16:    end for
17: end procedure
```

Algorithm 4 GPU-based algorithm to implement the SpMV (veCSC) operation of Algorithm 1 (lines 19 and 37) with the sparse adjacency matrix in the CSC format.

```
1: Input: $\boldsymbol{x}$,$\boldsymbol{CP_A}$,$\boldsymbol{row_A}$
2: Output: $\boldsymbol{y}$
3: procedure veCSC-SpMV-Kernel($\boldsymbol{x}$,$\boldsymbol{CP_A}$,$\boldsymbol{row_A}$,$\boldsymbol{y}$)
4:     $thread_{id} \leftarrow threadIdx.x + blockIdx.x * blockDim.x$
5:     $threadLane_{id} \leftarrow thread_{id} \& (threadsPerWarp - 1)$
6:     $warp_{id} \leftarrow thread_{id}/threadsPerWarp$
7:     while $col < n$ do
8:         if $\sigma(col) == 0$ then
9:             $start \leftarrow CP_A(warp_{id})$
10:            $end \leftarrow CP_A(warp_{id} + threadLane_{id})$
11:            $sum \leftarrow 0$
12:            $icp \leftarrow start + threadLane_{id}$
13:            while $icp < end$ do
14:                $sum \leftarrow sum + y(row_A(icp))$
15:                $icp \leftarrow icp + threadsPerWarp$
16:            end while
17:            $offset \leftarrow threadsPerWarp/2$
18:            while $offset > 0$ do
19:                $sum \leftarrow sum + \text{shfl} - \text{sync}(mask, sum, offset)$
20:                $offset \leftarrow offset/2$
21:            end while
22:            if $threadLane_{id} == 0 \wedge sum > 0$ then
23:                $y(warp_{id}) \leftarrow sum$
24:            end if
25:        end if
26:        $col \leftarrow col + num - warps$
27:    end while
28: end procedure
```

Our experiments showed that for some medium size graphs (see Table 1), the best performance was obtained with Algorithm 3, which implements the sequential version of the SpMV operations on Algorithm 1, with the sparse adjacency matrix in the CSC format. Algorithm 3 implements the mask operation (line 5) by computing the components of the vector $\boldsymbol{y}$ only when the corresponding component of the $\boldsymbol{\sigma}$ vector is equal to 0, ensuring that only the new discovered shortest paths are added to $\boldsymbol{\sigma}$ (line 25 of Algorithm 1). The sparsity of vector $\boldsymbol{x}$ is used on line 12 when the vector $\boldsymbol{y}$ is updated only when the variable *sum* is greater than zero. The straightforward parallelization of Algorithm 3, known as CSC-scalar (scCSC), with a GPU kernel, assigns one thread per vertex. In this paper, the acronym TurboBC-scCSC designated the BC algorithm using the scCSC algorithm for the SpMV operation.

Our experiments for irregular graphs, showed that both the TurboBC-scCOOC and the TurboBC-scCSC algorithms resulted in poor performance due to uncoalesced memory access and warp divergence. In order to improve the performance of the SpMV operation for irregular graphs, we implemented the CSC-vector(veCSC) algorithm shown in Algorithm 4, which is similar to the CSR-vector algorithm proposed in [3]. The veCSC algorithm assigns a warp for vertex. This algorithm incorporates the warp shuffle instruction (lines 18-21) to reduce the local sums by the threads in the warp without using shared memory. The first thread in the warp outputs the final result (lines 22-24). The veCSC algorithm solves the problems of no coalesced memory access and warp divergence of the scalar algorithms when applied to irregular graphs. The best performance of the veCSC algorithm is obtained for irregular graphs, on which the warp divergence is minimized. The acronym TurboBC-veCSC designated the BC algorithm using the veCSC algorithm for the SpMV operation.

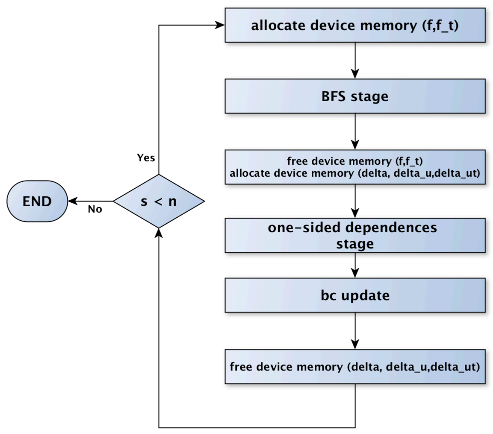
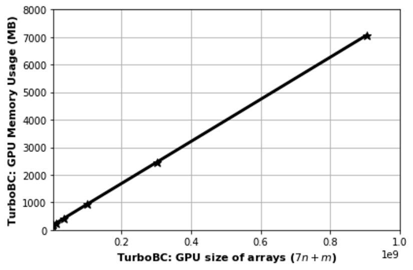

allocate device memory (f,f_t)
BFS stage
free device memory (f,f_t)
allocate device memory (delta, delta_u,delta_ut)
one-sided dependences stage
bc update
free device memory (delta, delta_u,delta_ut)
s < n
Yes
No
END

Figure 2: Pipeline for the CUDA implementation of Algorithm 1.

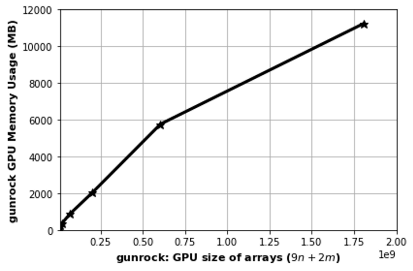
3.4 CUDA implementation of the BC algorithm

We designed and implemented Algorithm 1 using the pipeline shown in Figure 2. The BFS stage of Algorithm 1 was implemented using two kernels, the first kernel initializes the $\boldsymbol{f}$ and $\boldsymbol{\sigma}$ vectors (lines 15 to 18) and executes the SpMV ($\boldsymbol{f_t} \leftarrow \boldsymbol{f}A$) operation (line 19), the second kernel computes the additional functions of this stage. The computation of the one-sided dependences vector, $\boldsymbol{\delta}$, was implemented using three kernels, the first kernel updates the vector $\boldsymbol{\delta_u}$ (lines 34-36), the second kernel computes the SpMV operation (line 37), and the third kernel updates the vector $\boldsymbol{\delta}$ (lines 38-40). One additional kernel updates the vector $\boldsymbol{bc}$ (lines 43 to 47). This implementation increased the performance of the algorithm by reducing the overhead due to the sequential execution of too many kernels on the GPU.

Our experiments showed that the performance of Algorithm 1 increased when the SpMV operation of the BFS stage was performed over integer data types for the $\boldsymbol{f}$ and $\boldsymbol{f_t}$ vectors. Hence, in order to minimize the memory-footprint on the GPU due to the storage of auxiliary vectors, the deallocation of the device memory for the vectors $\boldsymbol{f}$ and $\boldsymbol{f_t}$ is followed by the allocation of device memory to the float type vectors $\boldsymbol{\delta}, \boldsymbol{\delta_u}$, and $\boldsymbol{\delta_{ut}}$. The SpMV operation with integer data types was up to 2.7x faster than the same operation with float data types, with a very small overhead due to the allocation and deallocation operations of the device memory.



a) GPU memory upper bound TurboBC



b) GPU memory upper bound gunrock

Figure 3: GPU memory upper bounds obtained with the TurboBC-veCSC algorithm compared with the values obtained by the gunrock BC algorithms for the computation of BC/vertex in the mycielski group of irregular graphs included in Table 3.

In order to quantify the reduction in the memory footprint of the TurboBC algorithms, Figure 4 shows the data flow for the BC algorithms in the gunrock library and for our TurboBC algorithms. The host (CPU) arrays (yellow) are shown as inputs to the memory transfer block, the auxiliary arrays (green) are used by the GPU to

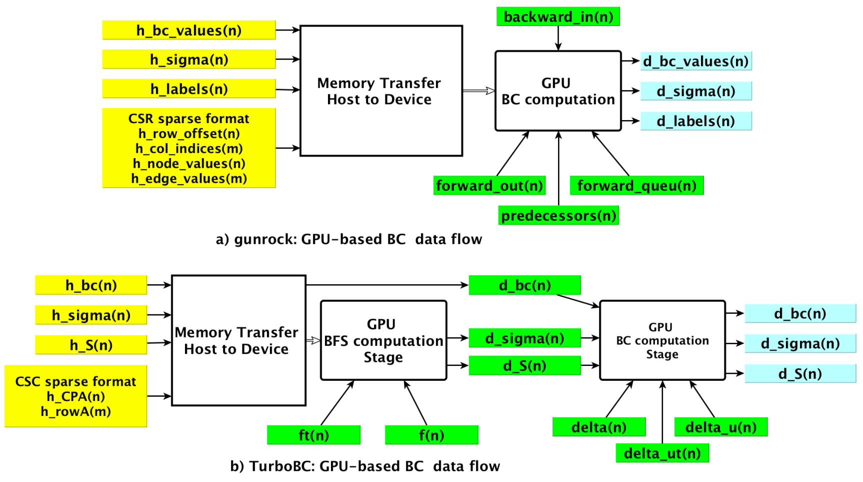


Figure 4: Data flow for the GPU-based BC algorithms implemented in the gunrock library and in TurboBC.

compute the BC which results in the output arrays (blue). The size of each array is given, with n as the number of vertices and m as the number of edges. We assumed that a lower bound for the global memory required by the GPU during the BC computation was proportional to the total size of the arrays required by this computation, which in the case of the gunrock library is equal to $9n + 2m$, and for the TurboBC is equal to $7n + m$ for the BC computation stage. Figure 3 a) and b) show the expected linear relationship between the GPU memory usage, considered an experimental GPU memory upper bound, and the total size of the arrays for the computation of BC with TurboBC and with the gunrock libray, respectively. Our experimental results also showed that the reduction, proportional to $2n + m$, in the GPU global memory requirements of the TurboBC algorithms, illustrated in Figure 5 a), allowed the computation of the BC for the relatively big graphs given in Table 4, while the BC algorithms in the gunrock library ran out of memory for these type of graphs, more details about these computations are given in Sections 4.2 and 4.3.

4 EXPERIMENTAL RESULTS

The experiments presented in this section were designed to assess the performance of our TurboBC algorithms by comparing them to the benchmark parallel BC algorithms available in the state-of-the-art GPU-based gunrock [21] and CPU-based, shared memory, ligra [20] libraries. We also compared the performance of the TurboBC algorithms with the performance of our implementation of the sequential version of Algorithm 1 with the sparse adjacency matrix in the CSC format.

Our benchmark of thirty-three graphs used in the experiments were represented by sparse adjacency matrices selected from the SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection) [11], and from the Stanford Large Network Dataset Collection [12]. The selected adjacency matrices represented eighteen undirected and fifteen directed graphs, covering a wide range of vertices $[28 \times 10^3, 214 \times 10^6]$ and edges $[171 \times 10^3, 1950 \times 10^6]$. The parameters for the selected graphs are given in Tables 1,2, 3, and 4. The weighted graphs were considered unweighted graphs for all the experiments.

The average runtime (milliseconds) for each experiment was obtained by 50 trials per experiment. We used the sequential version of the BC algorithm to verify the results obtained from the TurboBC algorithms, only the correct results were accepted. For all the results presented in this section, we chose the TurboBC algorithm which showed the best performance for each graph. For the experiments on which the BC was computed for one vertex, the MTEPs (millions of transverse edges by second), achieved for the BC algorithms, were computed as the ratio m/t where m is the number of edges (thousands) and t is the average runtime (milliseconds). For the exact BC experiments on which the BC was computed for all the vertices in the graph, the MTEPs were computed as mn/t where n is the number of vertices (t in seconds, mn in millions).

All the experiments presented in this section were performed on a Linux server with Ubuntu operating system version 16.04.6, 22 Intel Xeon Gold 6152 processors, clock speed 2.1 GHz, and 125 GB of RAM. The GPU in this server was a NVIDIA Titan Xp, with 30 SM, 128 cores/SM, maximum clock rate of 1.58 GHz, 12196 MB of global memory, and CUDA version 10.1.243 with CUDA capability of 6.1.

4.1 Experimental results for regular graphs

This section summarizes the results of the experiments performed with the TurboBC algorithms to compute the BC of one vertex on twenty regular graphs, twelve of them directed graphs and the rest undirected graphs.

The number of vertices (n) and edges (m), the parameters (maximum, mean, standard deviation) of the degree (out-degree for directed graphs) distribution of the graphs, as well as the depth of the BFS tree (d), are given for each graph in Tables 1 and 2. These Tables also include the runtime, MTPEs and the speedup obtained by the TurboBC algorithms over the algorithms implemented on the gunrock (**(gunrock)x**) and ligra (**(ligra)x**) libraries, and over the sequential algorithm (**(sequential)x**). The symbol *OOM* means that

Table 1: Parameters and experimental runtime, MTEPs and speedup obtained with the TurboBC-scCSC algorithm over the sequential algorithm ((sequential)x) and over the gunrock ((gunrock)x) and the ligra ((ligra)x) libraries for the computation of the BC/vertex of a set of regular graphs.

File	n×10^3	m×10^3	degree(max/μ/σ)	d	scf	runtime	MTEPs	(sequential)x	(gunrock)x	(ligra)x
mark3j060sc(D)	28	171	44/6/4	42	10	2.1	82	11.5x	2.7x	2.2x
mark3j080sc(D)	37	228	44/6/4	52	10	2.8	82	9.8x	2.5x	1.5x
mark3j100sc(D)	46	285	44/6/4	62	10	3.5	82	11.4x	2.4x	1.5x
mark3j120sc(D)	55	343	44/6/4	72	10	4.4	78	12.9x	2.2x	1.6x
g7j140sc(D)	42	566	153/14/24	15	197	1.2	472	12.5x	1.9x	2.3x
g7j160sc(D)	47	657	153/14/24	16	208	1.4	469	13.3x	1.8x	2.6x
delaunayn15(U)	33	197	18/6/1	84	13	4.7	42	14.4x	2.4x	1.2x
delaunayn16(U)	66	393	17/6/1	110	14	7.1	55	25.3x	2.2x	1.9x
luxemb-osm(U)	115	239	6/2/0	1035	2	50.0	5	24.7x	2.3x	1.0x
internet(D)	125	207	138/2/4	21	1	1.5	138	37.8x	1.9x	2.0x

Table 2: Parameters, experimental runtime, MTEPs and speedup obtained with the TurboBC-scCOOC algorithm over the sequential algorithm ((sequential)x) and over the gunrock ((gunrock)x) and the ligra ((ligra)x) libraries, for the computation of the BC/vertex of a set of regular graphs.

File	n×10^3	m×10^3	degree(max/μ/σ)	d	scf	runtime(ms)	MTEPs	(sequential)x	(gunrock)x	(ligra)x
g7j180sc(D)	53	747	153/14/24	17	217	1.6	467	13.9x	1.7x	1.7x
g7j200sc(D)	59	838	153/14/25	18	224	1.7	493	14.6x	1.7x	1.8x
mark3j140sc(D)	64	400	44/6/4	82	10	5.3	76	13.2x	2.1x	1.2x
smallworld(U)	100	1000	17/10/1	9	61	1.0	1000	27.6x	1.5x	1.5x
ASIC-100ks(D)	99	579	206/6/6	33	3	2.7	215	25.7x	1.6x	1.7x
ASIC-680ks(D)	683	2329	210/3/4	31	2	6.6	353	43.9x	1.0x	1.5x
com-Youtube(U)	1135	5975	28754/5/51	14	8	9.7	616	48.4x	1.0x	2.8x
mawi-12345(U)	18571	38040	16×10^6/2/3806	10	2	74.8	509	33.6x	1.0x	3.6x
mawi-20000(U)	35991	74485	33×10^6/2/5414	11	2	143.0	521	33.9x	1.0x	3.4x
mawi-20030(U)	68863	143415	63×10^6/2/7597	12	2	261.4	549	32.3x	1.0x	3.2x

Table 3: Parameters and experimental runtime, MTEPs and speedup obtained with the TurboBC-veCSC algorithm algorithm over the sequential algorithm ((sequential)x) and over the gunrock ((gunrock)x) and the ligra ((ligra)x) libraries for the computation of the BC/vertex on a set of irregular graphs.

File	n×10^3	m×10^3	degree(max/μ/σ)	d	scf	runtime(ms)	MTEPs	(sequential)x	(gunrock)x	(ligra)x
mycielski15(U)	25	11111	12287/452/664	3	41166	1.7	6536	17.4x	1.2x	2.3x
mycielski16(U)	49	33383	24575/679/1078	3	82833	3.4	9819	26.6x	1.5x	3.4x
mycielski17(U)	98	100246	49151/1020/1747	3	166407	7.9	12689	34.6x	1.7x	4.4x
mycielski18(U)	197	300934	98303/1531/2817	3	333199	18.5	16267	45.8x	2.1x	5.1x
mycielski19(U)	393	903195	196607/2297/4530	3	651837	48.9	18470	53.1x	2.7x	5.2x
kron-logn18(U)	262	21166	49164/81/454	6	5846	8.7	2433	31.6x	0.9x	1.1x
kron-logn19(U)	524	43563	80676/83/541	6	6609	17.4	2504	44.7x	1.0x	0.9x
kron-logn20(U)	1049	89241	131505/85/641	6	7410	58.4	1528	34.0x	1.3x	1.0x
kron-logn21(U)	2097	182084	213906/87/756	6	8161	193.2	943	24.5x	1.1x	1.0x

the corresponding benchmark algorithm ran out-of-memory, and the symbol 1.9x in the column (**(gunrock)x**) means that TurboBC was 1.9x faster than the BC algorithms in the gunrock library.

The TurboBC-scCSC algorithm showed the best performance for the ten regular graphs in Table 1, obtaining up to 472 MTEPs, as well as a maximum of 37.8x and an average of 17.4x speedup over the sequential code, a maximum of 2.7x and an average of 2.2x speedup over the the BC algorithm available in the gunrock library, and a maximum of 2.6x and an average of 1.8x speedup over the BC algorithm available in the ligra library. The scale free metrics scf for these group of regular graphs varied in the range [1,208] and 80 % of the graphs had a value below 15 for this metric. The depth (d) of the BFS tree was below 100 for 80 % of the graphs in this group, and for these graphs, the TurboBC-scCSC algorithm obtained the maximum values of speedup and MTEPs.

For the ten regular graphs in Table 2, the TurboBC-scCOOC algorithm showed the best performance. This algorithm obtained up to 1000 MTEPs, a maximum of 48.4x and an average of 28.7x speedup over the sequential code, a maximum of 2.1x and average of 1.3x speedup over the the BC algorithm available in the gunrock library, and a maximum of 3.6x and an average of 2.2x speedup over the BC algorithm available in the ligra library. The scale free metrics scf for these group of regular graphs was in the range [2,224] and 80 % of the graphs have a value less than 100 for this metric. Our experiments also showed that for the last four graphs in Table 2 with vertices with a maximum degree much higher than the mean value, the TurboBC-scCOOC based on the COOC format had a better performance than the TurboBC algorithms based on the CSC format, and also than the corresponding algorithms in the ligra library, asserting that the COOC format results in scalar algorithms that are less affected for vertices with high degrees as compared to scalar algorithms based on the CSC format.

4.2 Experimental results for irregular graphs

This section summarizes the results of the experiments performed to compute the BC of one vertex on the nine irregular undirected

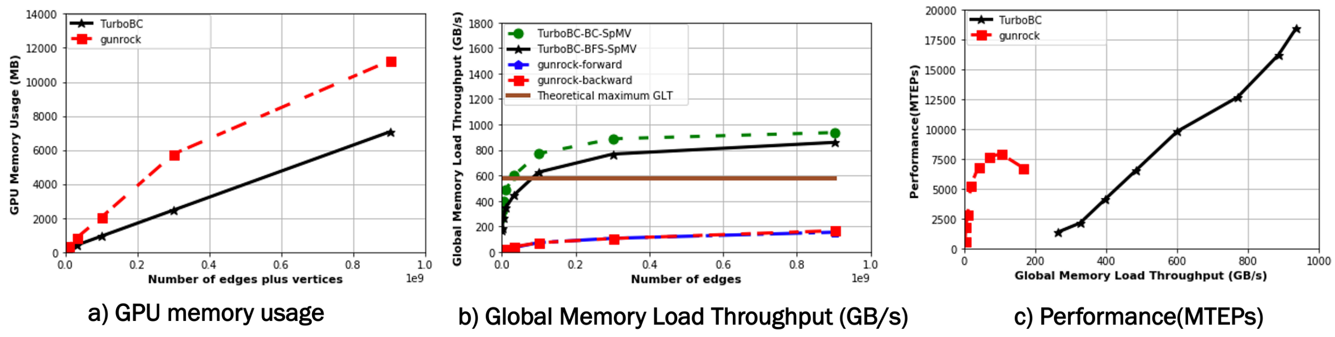


Figure 5: GPU memory usage, GPU memory upper bounds, Global Memory Load Throughput (GLT) and performance(MTEPs) obtained with the TurboBC-veCSC algorithm compared with the values obtained by the gunrock BC algorithms for the computation of BC/vertex in the mycielski group of irregular graphs included in Table 3.

Table 4: Parameters and experimental runtime, MTEPs and speedup obtained with the TurboBC algorithms over the sequential algorithm ((seq)x) and over the ligra ((ligra)x) library for the computation of the BC of a set of big graphs. The BC algorithm on gunrock ran out of memory for these graphs.

File	n×10^6	m×10^6	degree(max/μ/σ)	d	scf	runtime(s)	MTEPs	(sequential)x	(ligra)x
kmer-V1r(U)	214	465	8/2/1	324	2	14.3	33	94.5	0.9x
it-2004(D)	42	1151	9964/28/67	50	543	3.1	371	39.5	0.8x
GAP-twitter(D)	62	1469	3×10^6/24/1990	15	126	7.3	201	50.4	0.8x
sk-2005(D)	51	1950	12870/39/78	54	1262	6.8	287	30.5	0.7x

graphs given in Table 3. The scale free metrics scf for these group of graphs varied in the range [5846,651837], and as expected, the best performance on the computation of BC for these graphs was obtained by the TurboBC-veCSC algorithm. This algorithm obtained up to 18.5 GTEPs, as well as a maximum of 53.1x and an average of 34.7x speedup over the sequential code, a maximum of 2.7x and an average of 1.5x speedup over the the BC algorithm available in the gunrock library, and a maximum of 5.2x and an average of 2.7x speedup over the BC algorithm available in the ligra library. The TurboBC-veCSC algorithm obtained the maximum values of speedup and MTEPs for the mycielski group of graphs for which the depth (d) of the BFS tree was equal to 3.

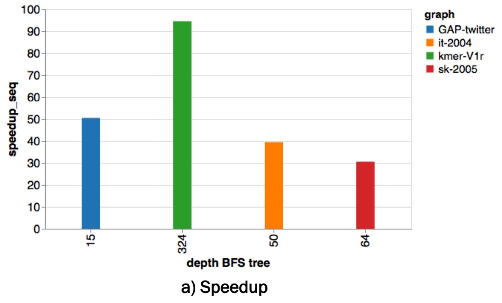
Figure 5a) compares the GPU memory usage by the TurboBC-veCSC algorithm and by the BC algorithms in the gunrock library during the computation of BC for the mycielski group of irregular graphs in Table 3. Since the space complexity of the Brandes' algorithm is O(m+n), Fig. 5a) shows that there is a linear relationship between the GPU memory usage and the sum of the number of vertices plus the number of edges of the mycielski graphs. Due to the strategy of reducing the memory footprint of the TurboBC algorithms, the memory usage of the gunrock library was up to 60 % higher than the memory usage of the TurboBC-veCSC algorithm.

The Global Memory Load Throughput (GLT) is a GPU metric that measures the rate at which the GPU global memory is accessed by an SM [7], Figure 5b) compares this metric obtained by the most important kernels of the TurboBC-veCSC algorithm, and by the kernels of the BC algorithm in the gunrock library. The theoretical maximum GLT achievable for the GPU (NVIDIA Titan Xp) used in our experiments was 575 GB/s, represented by the horizontal line in Figure 5b), the GLT obtained by the kernels in the gunrock library were substantially below this value, while the kernels in the TurboBC-veCSC algorithm obtained GLT values that were 60 % higher than the theoretical maximum GLT. Figure 5c) illustrates that the MTEPs, as function of the GLT metric, obtained by the TurboBC-veCSC algorithm were much higher than those obtained by the BC algorithms in the gunrock library.

In summary, the experimental results presented in Figure 5, showed that for a representative group of highly irregular big graphs, the TurboBC algorithms used memory more efficiently than the BC algorithms in the gunrock library.

4.3 Experimental results for the computation of BC for big graphs

Table 4 summarizes the results of the experiments performed for the computation of the BC/vertex of a set of of four big graphs with the TurboBC algorithms. The first graph of this set is a regular graph for which the TurboBC-scCSC algorithm showed the best performance, and the other three graphs are irregular directed graphs. The directed graph sk-2005 in Table 4 is the largest graph for which the TurboBC was computed with our available GPU. For the it-2004 irregular graph, the best performance was obtained with the TurboBC-scCOOC algorithm, for the other two irregular graphs the best performance was obtained with the TurboBC-veCSC algorithm, because the TurboBC-scCOOC algorithm run out of memory (OOM). For all the graphs on the set, the BC algorithm on the gunrock library ran out of memory (OOM), asserting our optimization strategy of reducing the memory footprint to design and implement our highly scalable TurboBC algorithms.



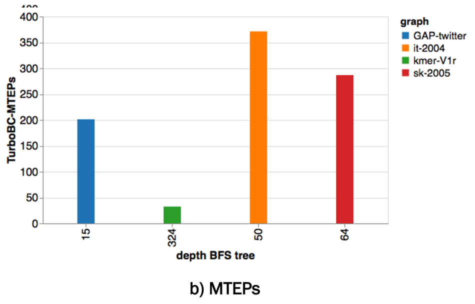


Figure 6: Experimental results for a) the speedup over the sequential algorithm and b)MTEPs obtained for our TurboBC algorithms in the computation of BC for the set of big graphs of Table 4.

The TurboBC algorithms obtained for this BC computation, up to 371 MTEPs, and a maximum of 94.5x and an average of 53.8x speedup over the sequential code. Since the BC algorithms in ligra used the CPU resources, specially the memory resources, effectively, there were up to 1.4x faster than the TurboBC algorithms for the computation of BC in these big graphs.

Figure 6a) shows that the greatest speedups of the TurboBC algorithms over the sequential BC algorithm were for the regular graph with the greatest value for the depth (d) of the BFS tree, and that the maximum values for the METPs were obtained with the TurboBC-veCSC algorithm applied to the irregular directed graphs, for which the depth (d) of the BFS tree was equal to or less than 50.

4.4 Experimental results for the exact BC computation of a set of graphs

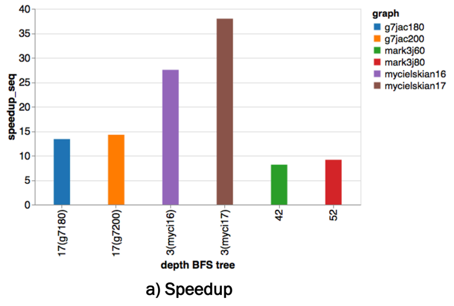
Table 5 summarizes the experimental results obtained by the TurboBC algorithms for the exact BC computation for all vertices of the set of six graphs. The first four are directed regular graphs, and the last two are undirected irregular graphs.

The parameters and the TurboBC algorithm used for these computations for the regular graphs are included in Table 2, and for the irregular graphs in Table 3. The parameter $\mathbf{n} \times \mathbf{m}$ is included in Table 5, because the MTEPs for the exact BC computation are computed as the ratio between the number of edges times the number of vertices (millions) and the average runtime (seconds). The TurboBC algorithms obtained for this exact BC computation, up to 13.8 GTEPs and a maximum of 38.0x and an average of 18.4x speedup over the sequential code. The results in Table 5, also show that both the speedup and the MTEPs increased with the size of the graph, showing the high scalability of the TurboBC algorithms for this type of computation.

Table 5: Experimental runtime, MTEPs and speedup obtained with the TurboBC algorithms over the sequential algorithm ((seq.)x) for the computation of the exact BC of all vertices of a set of undirected and directed graphs.

File	d	n×m $\times 10^6$	runtime(s)	MTEPs	(seq.)x
mark3j60sc(D)	42	4694	49.3	95	8.2x
mark3j80sc(D)	52	8345	90.8	92	9.2x
g7j180sc(D)	17	39906	105.9	377	13.4x
g7j200sc(D)	17	49688	129.7	383	14.3x
mycielski16(U)	3	1639081	159.8	10257	27.5x
mycielski17(U)	3	9854152	715.2	13778	38.0x

Figure 7 shows that the maximum values for speedups and for MTEPs were obtained, for the graphs with the smaller values of the depth (d) of the corresponding BFS trees.



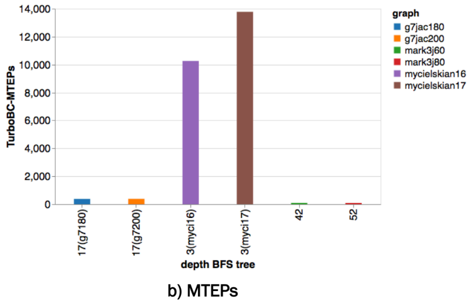


Figure 7: Experimental results for a) the speedup and b)the MTEPs of the exact computation of BC for the graphs given in Table 5.

5 RELATED WORK

The implementation of Brandes' algorithm on GPUs is an important area of research. One of the first implementations of this algorithm on GPU was the Jia et.al. [9] edge-parallel approach, followed by the gpu-fan package described in [19], which was based on an improved All-Pairs Shortest Path (APSP) algorithm. Several hybrid GPU-CPU and multiple GPU implementations are presented in [14, 16, 18]. The Brandes' BC algorithm have been implemented on high performance, parallel graphs processing libraries such as gunrock on the GPU [21], and the CPU-based, shared memory, ligra [20]. As far as we know, the BC algorithm in the language of linear algebra was first described on chapter 6 of reference [10], and implemented in the GraphBLAS library for CPUs [6], being our proposed TurboBC algorithms, the first GPU-based implementation of BC inspired in this linear algebra algorithm.

The gunrock BC algorithms use techniques such as BFS push-pull, that, as illustrated in Figure 4, require to store additional auxiliary arrays on the GPU global memory, increasing the space complexity of the algorithms and limiting the size of the graphs for which the BC can be computed with limited memory GPUs. Our approach for designing and implementing the algorithms in TurboBC differed from the gunrock approach, because we used memory efficient and highly scalable algorithms which were simpler and hence with less overhead. We also reduced the memory footprint of the TurboBC algorithms by using only one sparse storage format for each BC computation, by minimizing the number of auxiliary arrays on the device side, and by transferring to the GPU only one set of the arrays that store the indices of the non-zero values of the sparse adjacency matrices representing the graphs. This reduction in space complexity reduced the memory usage of the TurboBC algorithms, allowing the computation of the BC for graphs with higher number of vertices and edges than those computed by the gunrock library on the same GPU, and also increasing the performance of the TurboBC algorithms.

6 SUMMARY AND FUTURE WORK

In this paper, as far as we know, we designed and implemented TurboBC, the first memory efficient and highly scalable GPU-based set of BC algorithms in the language of linear algebra. The algorithms in TurboBC are applicable to unweighted, directed and undirected graphs represented by sparse adjacency matrices.

The design goals of the TurboBC algorithms were to reduce the GPU global memory footprint required by the algorithms, as well as to exploit the sparsity structure of the output and frontier vectors of the BFS stage. Our experiments showed that our TurboBC algorithms obtained more than 18 GTEPs (billions of transverse edges per second), and were on average 1.7x and 2.2x faster than the state-of-the-art algorithms implemented on the high performance, GPU-based, gunrock [21], and CPU-based, ligra [20] libraries, respectively. The experimental results also showed that by minimizing their memory footprint, the TurboBC algorithms were able to compute the BC of relatively big graphs, for which the gunrock algorithms ran out of memory.

Our future work will be focused on improving the performance of the algorithms in TurboBC, especially the performance of the algorithms computing the vector sparse matrix multiplication operations. Our goal will be to design and implement memory efficient and scalable GPU-based BC algorithms, with higher performance for big graphs than the state-of-the-art BC algorithms.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundations (NSF) under the Award Numbers CAREER OAC-1925960. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Science Foundation. We would also like to acknowledge the donation of a K-40c Tesla GPU and a TITAN Xp GPU from NVIDIA which was used for all the GPU-based experiments performed in this paper.

REFERENCES

[1] Oswaldo Artiles and Fahad Saeed. 2021. TurboBFS: GPU Based Breadth-First Search (BFS) Algorithms in the Language of Linear Algebra. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE (Ed.). IEEE, IEEE Publishing, New York, USA, 520–528.

[2] Alex Bavelas. 1948. A MATHEMATICAL MODEL FOR GROUP STRUCTURES. *Applied Anthropology* 7, 3 (1948), 16–30.

[3] Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication in CUDA.* Technical Report NVR-2008-004. NVIDIA.

[4] Bonnie Berger, Jian Peng, and Mona Singh. 2013. Computational solutions for omics data. *Nature Reviews Genetics* 14, 5 (2013), 333–346.

[5] Ulrik Brandes. 2008. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks* 30, 2 (2008), 136–145.

[6] Aydin Buluç, Tim Mattson, Scott McMillan, Jose Moreira, and Carl Yang. 2017. Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Press, New Orleans, Lousianna, 643–652.

[7] NVIDIA corporation. 2021. *CUDA C++ PROGRAMMING GUIDE.* Technical Report PG-02829-001-v11.3. NVIDIA.

[8] Linton C. Freeman. 1977. A Set of Measures of Centrality Based on Betweenness. *Sociometry* 40, 1 (March 1977), 35–41.

[9] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C. Hart. 2012. Chapter 2 - Edge v. Node Parallelism for Graph Centrality Metrics. In *GPU Computing Gems Jade Edition*. Morgan Kaufmann, Boston, 15–28.

[10] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[11] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The SuiteSparse Matrix Collection Website Interface. *Journal of Open Source Software* 4, 35 (2019), 1244.

[12] Jure Leskovec and Rok Sosic. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* 8, 1 (2016), 1–20.

[13] Lun Li, David Alderson, Reiko Tanaka, John C. Doyle, and Walter Willinger. 2005. *Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications (Extended Version).* Technical Report CIT-CDS-04-006. California Institute of Technology.

[14] 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*. IEEE Press, New Orleans, Louisana, 572–583.

[15] Tore Opsahl, Filip Agneessens, and John Skvoretz. 2010. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social Networks* 32, 3 (2010), 245–251.

[16] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and J. D. Owens. 2017. Multi-GPU Graph Analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Press, Orlando, Florida, 479–490.

[17] Mikail Rubinov and Olaf Sporns. 2010. Complex network measures of brain connectivity: uses and interpretations. *Neuroimage* 52, 3 (2010), 1059–1069.

[18] Ahmet Erdem Sariyuce, Kamer Kaya, Erik Saule, and Umit V. Catalyurek. 2013. Betweenness Centrality on GPUs and Heterogeneous Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, Houston, Texas, USA, 76–85.

[19] Zhiao Shi and Bing Zhang. 2011. Fast network centrality analysis using GPUs. *BMC Bioinformatics* 12, 1 (2011), 149.

[20] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Shenzhen, China, 135–146.

[21] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Barcelona, Spain, 1–12.

A ONLINE RESOURCES

The codes to implement the algorithms proposed in this paper are available at https://github.com/pcdslab.