2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-3577-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPSW52791.2021.00084

TurboBFS: GPU Based Breadth-First Search (BFS) Algorithms in the Language of Linear Algebra

Oswaldo Artiles, Life Senior Member, IEEE
School of Computing and Information Sciences
Florida International University
Miami, Florida
Email: {oarti001@fiu.edu}

Fahad Saeed, Senior Member, IEEE
School of Computing and Information Sciences
Florida International University
Miami, Florida
Email: {fsaeed@fiu.edu}

Abstract—**Graphs that are used for modeling of human brain, omics data, or social networks are huge, and manual inspection of these graph is impossible. A popular, and fundamental, method used for making sense of these large graphs is the well-known Breadth-First Search (BFS) algorithm. However, BFS suffers from large computational cost especially for big graphs of interest. More recently, the use of Graphics processing units (GPU) has been promising, but challenging because of limited global memory of GPU's, and irregular structures of real-world graphs. In this paper, we present a GPU based linear-algebraic formulation and implementation of BFS, called TurboBFS, that exhibits excellent scalability on unweighted, undirected or directed sparse graphs of arbitrary structure. We demonstrate that our algorithms obtain up to 40 GTEPs, and are on average 15.7x, 5.8x, and 1.8x faster than the other state-of-the-art algorithms implemented on the SuiteSparse:GraphBLAS, GraphBLAST, and gunrock libraries respectively. The codes to implement the algorithms proposed in this paper are available at https://github.com/pcdslab.**

Index Terms—**GPU, CUDA, graph parallel algorithms, BFS, linear algebra.**

I. Introduction

Graphs that are used for modeling of human brain [4], omics data [11], or social networks [12], are huge, making manual inspection of these graph practically impossible. A popular, and fundamental, method used for making sense of these large graphs is the well-known Breadth-First Search (BFS) algorithm with many interesting applications. For instance, the BFS algorithm is the first stage of the betweenness centrality (BC) algorithm due to Brandes [7], [13]. However, the high complexity of BFS algorithms is a severe bottleneck for numerous computational problems. Due to its importance, and high computational complexity with numerous applications; Graph 500 benchmark uses BFS as one of the algorithms for ranking supercomputers [2].

BFS algorithms, and its applications are both interesting, and challenging for Graphics Processing Units (GPU's) [16], because these algorithms (and architecture) have enough parallelism, but the data-access patterns are highly irregular. Other challenges for implementing BFS in a scalable fashion include limited global memory of the GPU's, the data-transfer PCIe bottleneck, and warp divergence on the GPU kernels. These challenges are primary reasons for limits on the size of the graph that can be processed in a scalable fashion by the available BFS algorithms, and to elicit limited-speedups on CPU-GPU architectures, being therefore an active area of research [17], [19], [20]

In this paper, we propose TurboBFS, a highly scalable GPU-based BFS algorithms in the language of linear algebra. The algorithms on TurboBFS are based on parallel optimizations selected to solve some of the problems associated to the challenges that we just discussed. We implemented a top-down BFS algorithm exploiting the sparsity of the frontier vector, which contains the number of shortest paths from the discovered vertices to the connected undiscovered vertices. Further exploitation of the sparsity is acquired by using the sparsity of the output vector, which contains the number of shortest paths from the root vertex to the discovered vertices. We also implemented the bottom up BFS algorithm presented in reference [5], as well as an algorithm that combined both approaches [6], which showed the best performance for some group of graphs. In order to optimize the use of the limited global memory of the GPU, we considered all the graphs unweighted, i.e, represented by Boolean sparse adjacency matrices [13], so that, the value arrays of the corresponding sparse formats were not stored. This result in reduction in the memory-footprint of the algorithms, resulting in substantial bandwidth utilization. Our algorithms were also designed to use only one type of sparse compressed format with the corresponding reduction in the memory footprint.

Contributions of the paper : The main contributions of the paper are:

1) We designed and implemented TurboBFS, a highly scalable GPU-based set of top-down and bottom-up BFS algorithms in the language of linear algebra. These algorithms are applicable to unweighted, directed and undirected graphs represented by sparse adjacency matrices in the Compressed Sparse Column (CSC) format, and the transpose of the Coordinate (COO) format, which were equally applicable to direct and undirected graphs. We considered all the graphs unweighted for which the adjacency matrices can be considered as Boolean matrices [13], so that the corresponding value arrays are not needed. This optimization resulted in substantial reduction of the GPU global memory footprint required by the algorithms, as well as an increased performance

due to reduction in the number of unnecessary floating operations. In our design, we also used the sparsity property of the frontier and output vectors to improve the performance of our BFS algorithms without using any additional memory.

2) A comprehensive experimental detail and results are presented to assess the performance of the GPU-based BFS algorithms in TurboBFS. Our BFS algorithms obtained up to 40 GTEPs (billions of transverse edges per second), and were on average 15.7x, 5.8x, and 1.8x faster than the state-of-the-art algorithms implemented on the SuiteSparse:GraphBLAS [1], GraphBLAST [20], and gunrock [19] libraries respectively.

Organization of the paper : The paper is organized as follows: Section II briefly describes the top-down and the bottom-up BFS algorithms. Section III presents the BFS algorithms for directed and undirected unweighted graphs in the language of linear algebra. Section IV is dedicated to the experimental results. Section V describes the related work, and the summary and future work are presented in Section VI.

II. Background

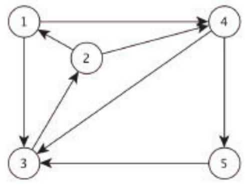
BFS algorithms : The breadth-first search (BFS) algorithm is applicable to any unweighted, directed or undirected graph $G = (V, E)$, where V is the finite set of vertices and E the set of edges. Any pair $(u, v) \in E$ implies that the vertices u and v in V are connected by an edge in G. A graph G is directed if E consists of ordered pairs, otherwise, G is undirected. Given a graph $G = (V, E)$ and a root vertex $r \in V$, the top-down BFS algorithm performs a systematic search of every vertex on E that is reachable from r. The algorithm computes the shortest path (smallest number of edges) from r to each reachable vertex, producing a BFS tree of the graph G. For connected graphs, the BFS tree is a spanning tree. On every step of the BFS algorithm, the frontier between discovered and undiscovered vertices is expanded. The algorithm discovers all the vertices at depth d, before discovering all the vertices at depth $d + 1$ on the BFS tree. On every step of the top-down BFS there are three sets of vertices. The first set, σ, contains the number of shortest paths from the root vertex to the discovered vertices, the second set, f, contains the number of shortest paths from the discovered vertices to the connected undiscovered vertices, and the third set are the undiscovered vertices, i.e., $V - \sigma - f$. The set f represents the frontier between the discovered and the undiscovered vertices. For a graph G with n vertices and m edges represented by a sparse adjacency matrix, the time complexity of the sequential BFS algorithm is $O(n^2)$ [9].

In every step of the top-down BFS algorithm, the vertices in f are like parents discovering their children among their neighbors. The step only finishes when each parent has searched for all the potential children. In the bottom-up BFS algorithm [5], the searching process is reversed, i. e., at every step, the undiscovered vertices are like children searching for parents, when a vertex finds a parent among its neighbors, the searching process finishes for that vertex. More details about our implementation of this algorithm are given in Section III.

III. BFS algorithms in the language of linear algebra for unweighted graphs.

This section presents our versions of the top-down and bottom-up BFS algorithms in the language of linear algebra for directed and undirected unweighted graphs. For the design and implementation of our algorithms, we used the Coordinate column COOC (transpose of the COO format) and the Compressed Sparse Column CSC sparse storage formats to represent the sparse adjacency matrices of the graphs. These formats are the best choice to compute the transpose sparse matrix vector multiplication ($y = A^T x$) performed in the top-down Algorithm 1, as well as the operations needed for the bottom-up Algorithm 6. Fig. 1 shows examples of these formats for a sparse adjacency matrix representing a directed, unweighted graph. For a $n \times n$ adjacency sparse matrix A with nnz non-zero elements representing unweighted graphs, the array row_A (size nnz) stores the corresponding row indices of these non-zero values. The array CP_A (size $n + 1$) stores the indices that start a column, the first element of this array is always equal to 1 and the last element equal to $nnz + 1$. The array row_A of the COOC format are identical to the corresponding arrays of the CSC format, while the array col_A (size nnz) stores the corresponding column indices of the non-zero values in A. In order to reduce the memory footprint and increase the performance of the TurboBFS algorithms, the arrays that stores the non-zero values of the sparse adjacency matrix of unweighted graphs were not used by our algorithms.

In this paper, we classified the graphs in two classes: *regular* graphs and *irregular* graphs. The regular graphs were those with a degree distribution with a regular pattern, i.e., with relatively low values of maximum, mean and variance, while irregular graphs were graphs having some vertices with maximum degree which are many orders of magnitude greater than their mean, and standard deviations relatively larger than those shown by regular graphs. Fig. 2 shows the differences between the degree distribution of these two classes of graphs. This Figure shows the relative histogram for the degree distribution of a regular graph, delaunay23, with a maximum, mean and standard deviation equal to 28, 6.0 and 1.0 respectively, as well as the relative histogram for the degree distribution of an irregular graph, mycielskian19, with a maximum, mean and standard deviation of their degree distribution equal to 196607, 2297 and 4530 respectively. The dispersion of the degree distribution for the irregular graph is as expected much greater than the corresponding dispersion for the regular graph. In our experiments we found that the depth d of the BFS tree for irregular graphs is much lower than the depth for regular graphs. A greater depth means that the runtime of the BFS algorithm increases, for example, for the delaunay23 graph, with 8.4×10^6 vertices and 50×10^6 edges, d was equal to 1213 and the runtime 2014 ms, while for the mycielskian19 graph, with 393215 vertices and 903×10^6 edges, d was equal to 3 and the runtime 31 ms. Therefore, the topology of the graphs which determines the depth of the BFS trees had a huge impact on the performance of the TurboBFS 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

$$\mathrm{row_A} = [2 \;\; 3 \;\; 1 \;\; 4 \;\; 5 \;\; 1 \;\; 2 \;\; 4]$$
$$\mathrm{CP_A} = [1 \;\; 2 \;\; 3 \;\; 6 \;\; 8 \;\; 9]$$

COOC

$$\mathrm{row_A} = [2 \;\; 3 \;\; 1 \;\; 4 \;\; 5 \;\; 1 \;\; 2 \;\; 4]$$
$$\mathrm{col_A} = [1 \;\; 2 \;\; 3 \;\; 3 \;\; 3 \;\; 4 \;\; 4 \;\; 5]$$

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

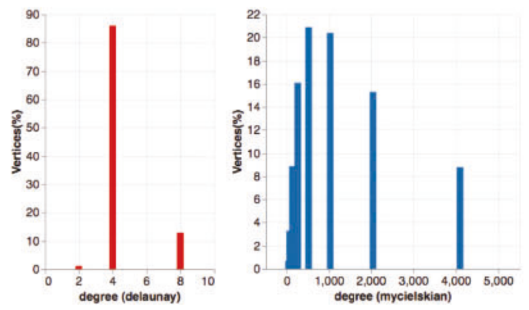


Fig. 2: Relative histogram of the degree distribution for the delaunay and mycielskian undirected group of graphs.

A. Top-Down BFS algorithm for undirected and directed unweighted graphs represented by sparse adjacency matrices in the COOC and CSC sparse storage formats.

Algorithm 1 represents the linear algebra formulation of the top-down BFS algorithm for a graphs $G = (V, E)$ with n vertices and nnz edges, represented by $n \times n$ sparse adjacency matrix A in the COOC format, with nnz non-zero elements. Algorithm 2 is the implementation of the same algorithm for the CSC format. These algorithms are inspired by the BFS algorithm described on chapter 6 of reference [13], where it is presented as a first step of the linear algebra version of the betweenness centrality algorithm [7]. The main innovation of our Algorithms 1 and 2 was the utilization of the sparsity of the *frontier* vector f and *output* vector σ to substantially improve the performance of the algorithms described in [13].

Algorithm 1 computes a top-down BFS from the root vertex r, with d representing the current depth of the discovered vertices. The final value of d is equal to the height of the BFS tree rooted on r. The output vector σ contains the number of shortest paths from the root vertex to the discovered vertices. The frontier vector f contains the number of shortest paths from the discovered vertices to the undiscovered vertices to which there is some edge. The while loop stops when the vector f is equal to 0, i.e., when all the vertices reachable from r have been discovered (lines 21 to 23). The vector f is updated by the sparse matrix-vector multiplication (SpMV) operation with the the transpose of the adjacency matrix (line 16), followed by a mask operation (lines 18 to 20) that updates on f the shortest paths to vertices no yet contained on the vector σ, guaranteeing that only the new discovered shortest paths are added to σ (lines 21 to 22).

Algorithm 1 Linear algebra formulation of the top-down BFS algorithm for an unweighted graph $G = (V, E)$ represented by a sparse adjacency matrix A in the COOC format.

```
1: Input: $A$.
2: Input: $r$.
3: Output: $\sigma(1....n)$
4: procedure TDBFS-LA-UG($A$,$\sigma$,$r$,$n$)
5:     $d \leftarrow 0$
6:     $c \leftarrow 1$
7:     $f \leftarrow 0$
8:     $\sigma \leftarrow 0$
9:     while $c > 0$ do
10:        $d \leftarrow d + 1$
11:        $c \leftarrow 0$
12:        if $d == 1$ then
13:            $f(r) \leftarrow 1$
14:            $\sigma(r) \leftarrow 1$
15:        end if
16:        $f_t \leftarrow A^T f$
17:        $f \leftarrow 0$
18:        if $\exists\sigma(i) == 0$ then
19:            $f(i) \leftarrow f_t(i)$
20:        end if
21:        if $\exists f(i)! = 0$ then
22:            $\sigma(i) \leftarrow \sigma(i) + f(i)$
23:            $c \leftarrow 1$
24:        end if
25:    end while
26: end procedure
```

Algorithm 2 represents the linear algebra formulation of the BFS algorithm for a graphs $G = (V, E)$ with the sparse adjacency matrix A in the CSC sparse storage format. This algorithm is similar to Algorithm 1, with the difference that due to the properties of the CSC format, the performance of the algorithm is improved by including the mask operation in the SpMV operation, as shown in Algorithm 4 .

B. Transposed sparse matrix-vector multiplication (SpMV) for the top-down and bottom-up BFS algorithms.

Our experimental results showed that the runtime of the SpMV operation on Algorithms 1 and 2 can be up to 90 % of their total runtime. Hence the overall performance of these algorithms was mainly determined by the performance of this operation. Algorithm 3 implements the sequential SpMV ($f_t \leftarrow A^T f$) operation of Algorithm 1. The parallel version of this algorithm, designated as scCOOC, assigns one thread per edge of the graph. The top-down BFS algorithm using the scCOOC algorithm for the SpMV operation was designated as TurboBFS-tdscCOOC.

Algorithm 4 implements the sequential version of the SpMV operation on Algorithm 2. The mask operation (line 2) is implemented by computing the components of the vector f_t

only when the corresponding component of the σ vector is equal to 0, ensuring that only the new discovered shortest paths are added to σ (lines 19 to 20 of Algorithm 2). The straightforward parallelization of Algorithm 4, known as CSC-scalar (scCSC), on a GPU kernel assigns one thread per vertex. In this paper, the acronym TurboBFS-tdscCSC designated the top-down BFS algorithm using the scCSC algorithm for the SpMV operation.

Algorithm 2 Linear algebra formulation of the top-down BFS algorithm for an unweighted graph G = (V, E) represented by sparse adjacency matrices A in the CSC format.

```
 1: Input: $A$.
 2: Input: $r$.
 3: Output: $\sigma(1....n)$
 4: procedure BFS-LA-UG($A$,$\sigma$,$r$,$n$)
 5:     $d \leftarrow 0$
 6:     $c \leftarrow 1$
 7:     $f \leftarrow 0$
 8:     $\sigma \leftarrow 0$
 9:     while $c > 0$ do
10:         $d \leftarrow d + 1$
11:         $c \leftarrow 0$
12:         $f_t \leftarrow 0$
13:         if $d == 1$ then
14:             $f(r) \leftarrow 1$
15:             $\sigma(r) \leftarrow 1$
16:         end if
17:         $f_t \leftarrow A^T f$
18:         $f \leftarrow f_t$
19:         if $\exists f(i)! = 0$ then
20:             $\sigma(i) \leftarrow \sigma(i) + f(i)$
21:             $c \leftarrow 1$
22:         end if
23:     end while
24: end procedure
```

Algorithm 3 Algorithm to implement the sequential $f_t \leftarrow A^T f$ operation of the BFS algorithm 1 with the sparse adjacency matrices in the COOC format.

```
1: for $k \rightarrow 1, nnz$ do
2:     if $f(row_A(k)) > 0$ then
3:         $\mathrm{f_t(col_A(k)) \leftarrow f_t(col_A(k)) + f(row_A(k))}$
4:     end if
5: end for
```

For irregular graphs, the scCSC kernel results 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 5, which is similar to the CSR-vector algorithm proposed in [3]. The veCSC algorithm assigns a warp per vertex. This algorithm incorporates the warp shuffle instruction (lines 18-22) 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 23-25). The veCSC algorithm solves the problems of no coalesced memory access and the warp divergence of the scCSC algorithm. The best performance of the veCSC algorithm is obtained for irregular graphs, on which the warp divergence is minimized. The acronym TurboBFS-tdveCSC designated the top-down BFS algorithm using the veCSC algorithm for the SpMV operation.

Algorithm 4 Algorithm (scCSC) to implement the sequential $f_t \leftarrow A^T f$ operation of the BFS algorithm 2 for sparse adjacency matrix in the CSC format .

```
 1: for $i \rightarrow 1, n$ do
 2:     if $\sigma(i) == 0$ then
 3:         $\mathrm{sum} \leftarrow 0$
 4:         $\mathrm{start} \leftarrow \mathrm{CP_A(i)}$
 5:         $\mathrm{end} \leftarrow \mathrm{CP_A(i+1) - 1}$
 6:         for $k \rightarrow start, end$ do
 7:             $\mathrm{sum} \leftarrow \mathrm{sum + f(row_A(k))}$
 8:         end for
 9:         if $sum > 0$ then
10:             $\mathrm{f_t(i) \leftarrow sum}$
11:         end if
12:     end if
13: end for
```

Algorithm 5 GPU-based algorithm (veCSC) to implement the sparse matrix-vector multiplication $f_t \leftarrow A^T f$ with the sparse adjacency matrix in the CSC format.

```
 1: Input: $CP_A$,$row_A$,$f$
 2: Output: $f_t$
 3: procedure VECSC-MVSP-KERNEL($CP_A$,$row_A$,$f$)
 4:     $\mathrm{thread_{Id} \leftarrow threadIdx.x + blockIdx.x * blockDim.x}$
 5:     $\mathrm{threadLane_{Id} \leftarrow thread_{Id}\ \&\ (threadsPerWarp - 1)}$
 6:     $\mathrm{warp_{Id} \leftarrow thread_{Id}/threadsPerWarp}$
 7:     $\mathrm{col \leftarrow warp_{Id}}$
 8:     while $col < n$ do
 9:         if $\sigma(col) == 0$ then
10:             $\mathrm{start \leftarrow CP_A(col)}$
11:             $\mathrm{end \leftarrow CP_A(col + threadLane_{Id})}$
12:             $\mathrm{sum \leftarrow 0}$
13:             $\mathrm{icp \leftarrow start + threadLane_{Id}}$
14:             while $icp < end$ do
15:                 $\mathrm{sum \leftarrow sum + f(row_A(icp))}$
16:                 $\mathrm{icp \leftarrow icp + threadsPerWarp}$
17:             end while
18:             $\mathrm{offset \leftarrow threadsPerWarp/2}$
19:             while $offset > 0$ do
20:                 $\mathrm{sum \leftarrow sum + shfl - down - sync(offset)}$
21:                 $\mathrm{offset \leftarrow offset/2}$
22:             end while
23:             if $threadLane_{id} == 0$ then
24:                 $\mathrm{f_t(warp_{Id}) \leftarrow sum}$
25:             end if
26:         end if
27:         $\mathrm{col \leftarrow col + numWarp}$
28:     end while
29: end procedure
```

C. Bottom-Up BFS algorithm for undirected and directed unweighted graphs represented by sparse adjacency matrices in the CSC sparse storage format.

Algorithm 6 computes a bottom-up BFS from the root vertex r, with d representing the current depth of the discovered vertices. This algorithm is the linear algebra version of the algorithm described in reference [5]. The final value of d is equal to the height of the BFS tree rooted on r. The output vector S contains the level d at which each vertex is discovered. If the vertex v is undiscovered (line 12), the searching for the parent of v starts at line 15 by searching all the incidents vertices to v, when a parent is found (line 16), the corresponding element of the matrix $\mathrm{S(v)}$ is updated with the next to the current level $(d+1)$. After discovering the parent of v, the searching process is completed with the break instruction on line 17. This early termination of the searching process is the main difference with the top-down BFS Algorithms 1 and 2 on which all the potential children have to be checked on each searching step of the algorithm. The straightforward parallelization of Algorithm 6 by a GPU kernel assigns one thread per vertex, we used the acronym TurboBFS-busc to designate this algorithm.

Algorithm 6 Linear algebra formulation of the bottom-up BFS algorithm for an unweighted graph G = (V, E) represented by sparse adjacency matrices A in the CSC format.

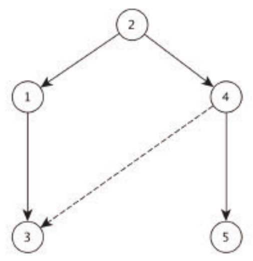
```
1: Input: $A$.
2: Input: $r$.
3: Output: $S(1....n)$
4: procedure BUBFS-LA-UG($A$,$S$,$r$,$n$)
5:     $d \leftarrow 0$
6:     $c \leftarrow 1$
7:     $S \leftarrow -1$
8:     $S(r) \leftarrow d$
9:     while $c > 0$ do
10:        $c \leftarrow 0$
11:        for $v \rightarrow 1, n$ do
12:            if $S(v) == -1$ then
13:                $k \leftarrow CP_A(v)$
14:                $end \leftarrow CP_A(v+1) - 1$
15:                while $k > end$ do
16:                    if $S(I(k)) == d$ then
17:                        $S(v) \leftarrow d+1$
18:                        $c \leftarrow 1$
19:                        $break$
20:                    end if
21:                    $k \leftarrow k+1$
22:                end while
23:            end if
24:        end for
25:        $d \leftarrow d+1$
26:    end while
27: end procedure
```

For irregular graphs, the TurboBFS-busc kernel results in poor performance due to uncoalesced memory access and warp divergence. In order to improve the performance of Algorithm 6 for irregular graphs, we implemented the TurboBFS-buve Algorithm 7, which is a simplified version of Algorithm 5 and where a warp is assigned to each vertex. The TurboBFS-buve algorithm solves the problems of no coalesced memory access and the warp divergence of the TurboBFS-busc algorithm for irregular graphs.

The bottom-up BFS algorithm has the best performance when a large fraction of the vertices are in the frontier [5]. At the beginning of the BFS search, the frontier vector is sparse and the top-down BFS is more efficient than the bottom-up approach. Hence, to yield the best performance, both algorithms can be combined, running the top-down BFS at the beginning of the process, and when the frontier vector becomes dense to switch to the bottom-up algorithm, in a direction optimizing BFS algorithm [6]. We designed and implemented a combined BFS algorithm on which the searching process starts with the top-down Algorithm 2, and then when the frontier vector f becomes dense, the searching process continues with the bottom-up Algorithm 6. We use the heuristics proposed in reference [6], to switch from the top-down to the bottom-up algorithm when the frontier vector has a minimum of 10% of nonzero elements. We used the acronym TurboBFS-tdbu to designate this combined algorithm.

Algorithm 7 TurboBFS-buve:GPU-based implementation of Algorithm 6, using one warp per vertex.

```
1: Input: $CP_A$,$row_A$,$S$,$d$
2: Output: $S$
3: procedure VECSC-MVSP-KERNEL($CP_A$,$row_A$,$S$,$d$),
4:     $\mathbf{thread_{Id}} \leftarrow \mathbf{threadIdx.x} + \mathbf{blockIdx.x} * \mathbf{blockDim.x}$
5:     $\mathbf{threadLane_{Id}} \leftarrow \mathbf{thread_{Id}}\ \&\ (\mathbf{threadsPerWarp} - 1)$
6:     $\mathbf{warp_{Id}} \leftarrow \mathbf{thread_{Id}/threadsPerWarp}$
7:     $\mathbf{col} \leftarrow \mathbf{warp_{Id}}$
8:     $\mathbf{break} \leftarrow \mathbf{0}$
9:     while $!break$ and $col < n$ and $S(col) == -1$ do
10:        $\mathbf{start} \leftarrow \mathbf{CP_A(col)}$
11:        $\mathbf{end} \leftarrow \mathbf{CP_A(col + threadLane_{Id})}$
12:        $\mathbf{icp} \leftarrow \mathbf{start + threadLane_{Id}}$
13:        while $!break$ and $icp < end$ do
14:            if $S(row_A(icp)) == d$ then
15:                $\mathbf{S(col) \leftarrow d}$
16:                $c \leftarrow \mathbf{1}$
17:                $\mathbf{break} \leftarrow \mathbf{1}$
18:            end if
19:            $\mathbf{icp} \leftarrow \mathbf{icp + threadsPerWarp}$
20:        end while
21:        $\mathbf{col} \leftarrow \mathbf{col + numWarp}$
22:    end while
23: end procedure
```



f	0	1	0	0	0	
σ	0	1	0	0	0	
thread	**1**	**2**	**3**	**4**	**5**	**d**
$f_t \leftarrow fA$	1	0	0	1	0	**1**
$f \leftarrow f_t$	1	0	0	1	0	**1**
$\sigma \leftarrow \sigma + f$	1	1	0	1	0	**1**
$f_t \leftarrow fA$	0	0	2	0	1	**2**
$f \leftarrow f_t$	0	0	2	0	0	**2**
$\sigma \leftarrow \sigma + f$	1	1	2	1	1	**2**
$f_t \leftarrow fA$	0	0	0	0	0	**3**
$f \leftarrow f_t$	0	0	0	0	0	**3**

Fig. 3: Example of computation of the BFS for the graph in Fig. 1 using the TurboBFS-scCSC version of Algorithm 2.

D. CUDA implementation of the BFS algorithms

We designed and implemented Algorithm 1 and Algorithm 2 with only two kernels on the GPU. The first kernel initializes the f and σ vectors and executes the SpMV($\boldsymbol{f_t} \leftarrow \boldsymbol{A^T f}$)

operation, the second kernel computes the additional functions of the algorithms. This implementation increased the performance of the algorithm by reducing the overhead due to the sequential execution of more than two kernels on the GPU. The implementation of the bottom-up BFS Algorithm 6 used only one GPU kernel.

Fig. 3 shows the computation of the top-down BFS for the root vertex 2 of the graph in the example of Fig. 1, using the TurboBFS-tdscCSC version of Algorithm 2. The example assigns one thread per vertex, and shows the values assigned to the vectors σ and f on each one of the two steps of the BFS computation.

IV. Experimental results

The experiments presented in this section were designed to assess the performance of our TurboBFS algorithms by comparing it to the performance of the benchmark algorithms available in the SuiteSparse:GraphBLAS library [1], and in the GPU based GraphBLAST [20], and gunrock [19] libraries.

Our benchmark of sixty-nine graphs used in the experiments are represented by sparse adjacency matrices selected from the SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection) [10], [14], some of these graphs are also in the Stanford Large Network Dataset Collection [15]. The selected adjacency matrices represent thirty-nine undirected and thirty directed graphs, with up to 1900×10^6 edges and up to 214×10^6 vertices. The parameters for each graph are given in the Tables I, II, and III. The weighted graphs were considered unweighted graphs for all the experiments.

The average running time (milliseconds) for each experiment was obtained by 50 trials per experiment. The MTEPs (millions of transverse edges by second) achieved for each BFS algorithm were computed as the ratio between the number of edges (thousands of edges) and the average running time (milliseconds). We also implemented a sequential BFS algorithm to verify the results obtained from the GPU-based algorithms, only the correct results were accepted. For all the results presented in this section, we chose the TurboBFS algorithm with the best performance.

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.

A. Experimental results for regular graphs

This section summarizes the results of the experiments performed for the computation of BFS on thirty-eight regular graphs, nineteen of them direct graphs and the rest undirected graphs. The number of vertices and edges, as well as the parameters (maximum, mean, standard deviation) of the degree (out-degree for directed graphs) distribution of the graphs, are given for each graph in Table I. This Table also include the MTPEs and the speedup of the TurboBFS algorithms over the algorithms implemented on the GraphBLAST ((**GBLAST**)**x**), gunrock ((**gunrock**)**x**), and SuiteSparse:GraphBLAS ((**GBLAS**)**x**) libraries. The symbol OOM mean that the corresponding benchmark algorithm ran out-of-memory, and the symbol 2.2x in the column ((**gunrock**)**x**) means that TurboBFS was 2.2x faster then the BFS algorithms in the gunrock libray.

The TurboBFS algorithms obtained up to 2000 MTEPs, and were on average 7.4x, 2.0x, and 11.7x faster than the BFS algorithms available on the GraphBLAST, gunrock, and SuiteSparse:GraphBLAS libraries respectively. The top-down TurboBC-tdscCSC algorithm obtained the best performance for twenty-one (55 %) of the graphs, and the bottom-up TurboBFS-busc algorithm showed the best performance for fifteen (40 %) of the graphs in this group.

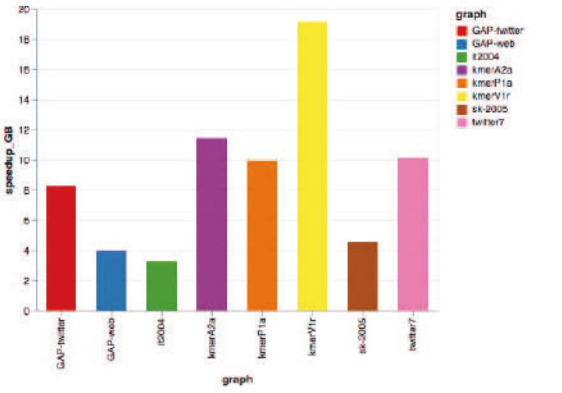


Fig. 4: Experimental results for the speedup obtained by the TurboBFS algorithms in the computation of BC for the set of big graphs of Table III.

B. Experimental results for irregular graphs

This section presents the results of the experiments performed for the computation of BFS on twenty-three irregular graphs, six of them direct graphs and seventeen undirected graphs.The parameters of the graphs as well as the speedup obtained with the TurboBFS algorithms are summarized in Table II. The TurboBFS algorithms obtained up to 40 GTEPs, and were on average 3.0x, 1.3x, and 22.4x faster than the BFS algorithms available on the GraphBLAST, gunrock, and SuiteSparse:GraphBLAS libraries respectively.

The top-down TurboBFS-tdveCSC algorithm obtained the best performance for seven (30 %), the bottom-up TurboBFS-buve algorithm for five (21.7 %), and the combined top-down bottom-up TurboBFS-tdbu algorithm for six (26.1 %) of the irregular graphs in Table II. The top-down TurboBFS-tdscCOOC algorithm obtained the best performance for five (21.7 %) of these irregular graphs, including the most irregular graphs, the mawi graphs, in the group, showing that the COOC format is the most suitable format for these highly irregular graphs.

TABLE I: Parameters and experimental MTEPs and speedup over the GraphBLAST ((**GBLAST**)**x**), gunrock ((**gunrock**)**x**), and SuiteSparse:GraphBLAS ((**GBLAS**)**x**) libraries, obtained with the TurboBFS algorithms for the computation of BFS for a set of regular graphs.

File	V×10^3	E×10^3	degree(max/μ/σ)	d	MTEPs	(GBLAST)x	(gunrock)x	(GBLAS)x
g7jac100sc(D)	30	385	153/13/22	14	769	8.4x	2.2x	10.4x
g7jac120sc(D)	36	475	153/13/23	14	792	7.0x	2.0x	8.3x
g7jac140sc(D)	42	566	153/14/24	15	1132	9.0x	2.4x	15.6x
g7jac160sc(D)	47	657	153/14/24	16	1094	8.0x	2.5x	13.3x
g7jac180sc(D)	53.	747	153/14/24	17	1068	7.4x	2.3x	13.4x
g7jac200sc(D)	59	838	153/14/25	17	1047	6.2x	2.4x	10.6x
cit-HepPh(D)	35	422	411/12/15	33	703	22.7x	4.2x	13.3x
email-Enron(U)	37	368	1383/10/36	10	613	4.5x	1.7x	6.3x
delaunayn15(U)	33	197	18/6/2	84	109	9.0x	3.1x	8.5x
delaunayn16(U)	66	393	17/6/2	110	164	8.9x	3.4x	11.7x
delaunayn17(U)	131	786	17/6/1	157	161	7.6x	2.2x	15.9x
delaunayn18(U)	262	1573	21/6/1	197	225	9.1x	2.0x	12.7x
delaunayn19(U)	524	3146	21/6/1	306	115	3.6x	1.0x	9.2x
astro-ph(U)	17	243	360/15/21	10	808	15.0x	2.7x	12.7x
ri2010(U)	25	126	44/5/3	66	90	11.1x	3.2x	7.4x
me2010(U)	70	336	73/5/3	108	112	9.4x	2.5x	8.3x
az2010(U)	242	1196	137/5/4	128	150	4.1x	1.3x	9.4x
nc2010(U)	289	1417	83/5/3	207	104	4.6x	1.2x	9.1x
fl2010(U)	484	2346	177/5/4	151	165	3.8x	0.9x	10.6x
ca2010(U)	710	3489	141/5/3	216	126	2.9x	0.8x	8.0x
enron(D)	69	276	1392/4/28	8	690	8.8x	2.0x	13.0x
Wordnet3(D)	83	133	64/2/2	20	190	10.6x	2.3x	6.6x
ASIC-100ks(D)	99	579	206/6/6	33	482	9.0x	2.7x	14.2x
ASIC-320ks(D)	322	1828	412/6/8	31	870	7.6x	1.8x	17.1x
ASIC-680ks(D)	683	2329	210/3/4	31	776	5.1x	1.4x	21.7x
smallworld(U)	100	1000	17/10/1	9	2000	8.0x	2.4x	24.2x
luxemb-osm(U)	115	239	6/2/1	1035	20	16.8x	6.7x	3.5x
netherl-osm(U)	2217	4883	7/2/1	1796	26	2.5x	0.8x	1.9x
internet(D)	125	207	138/2/4	21	345	8.3x	3.2x	29.2x
amazon0302(D)	262	1235	5/5/1	72	363	6.8x	2.1x	11.8x
amazon0312(D)	401	3200	10/8/3	45	593	3.5x	0.9x	9.1x
amazon0601(D)	403	3387	1/8/3	36	847	3.9x	1.0x	12.0x
amazon0505(D)	410	3357	10/8/3	36	839	3.9x	1.0x	13.3x
amazon-2008(D)	735	5158	10/7/4	32	992	2.9x	0.8x	14.2x
web-NtDame(D)	326	1497	3455/5/22	52	454	9.5x	1.6x	8.5x
roadNet-PA(U)	1091	3084	9/3/1	542	71	4.3x	1.0x	9.5x
roadNet-TX(U)	1393	3843	12/31	723	62	4.0x	0.9x	8.8x
roadNet-CA(U)	1971	5533	12/3/1	555	87	3.6x	1.0x	10.1x

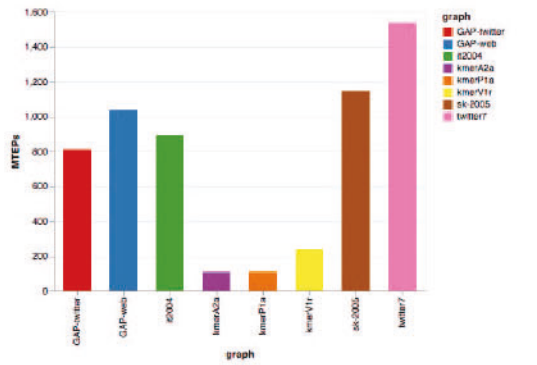


Fig. 5: Experimental results for the MTEPs obtained by the TurboBFS algorithms in the computation of BFS for the set of big graphs of Table III.

C. Experimental results for the computation of BFS for big graphs

Table III summarizes the results of the experiments performed for the computation of the BFS of a set of eight relatively big graphs with the TurboBFS algorithms. The directed graph sk-2005 in the Table is the largest graph for which the BFS was computed with our available GPU. The first three graphs of this set are regular graphs for which the bottom-up TurboBFS-busc algorithm showed the best performance, and the other five graphs are irregular graphs for which the best performance was obtained with the combined top-down and bottom-up TurboBFS-tdbu algorithm. For all the graphs on the set, the BFS algorithms on the gunrock and GraphBLAST libraries ran out of memory (OOM), asserting our optimization strategy of reducing the memory footprint to design and implement our highly scalable TurboBFS algorithms.

Fig. 4 shows that the greatest speedups of the TurboBC algorithms were obtained for the regular graphs. This Figure also shows that the maximum speedup of the TurboBFS-tdbu algorithm over the BFS algorithm on the GraphBLAS library was obtained for twitter7, the most irregular graph in the group with the smallest value for the depth (d) of the BFS tree. Fig. 5 shows that the largest value for the MTEPs was obtained for the twitter7 graph, while the smallest values for the METPs were obtained for the regular graphs which had the greatest

TABLE II: Parameters and experimental MTEPs and speedup over the GraphBLAST ((**GBLAST**)**x**), gunrock ((**gunrock**)**x**), and SuiteSparse:GraphBLAS ((**GBLAS**)**x**) libraries, obtained with the TurboBFS algorithms for the a set of irregular graphs.

File	$V \times 10^3$	$E \times 10^3$	degree($\max/\mu/\sigma$)	d	MTEPs	(GBLAST)x	(gunrock)x	(GBLAS)x
mycielski12(U)	3	407	$1535/133/149$	3	2036	5.5x	2.2x	7.5x
mycielski13(U)	6	1228	$3071/200/247.0$	3	4092	3.9x	1.8x	12.5x
mycielski14(U)	12	3696	$6143/301/407$	3	6159	2.0x	1.5x	23.3x
mycielski15(U)	24	11111	$12287/452/664$	3	11111	1.3x	1.7x	30.9x
mycielski16(U)	49	33383	$24575/679/1080$	3	16691	1.1x	1.5x	36.2x
mycielski17(U)	98	100246	$49151/1020/1747$	3	22277	0.8x	1.5x	42.2x
mycielski18(U)	196.6	300934	$98303/1531/2817$	3	31347	0.8x	1.7x	48.0x
mycielski19(U)	393	903195	$196607/2297/4530$	3	39778	OOM	1.8x	57.3x
EAT-SR(D)	23.2	326	$78/14/20$	7	1085	6.3x	2.7x	16.7x
kron-logn16(U)	66	4913	$17999/75/313$	6	4466	2.1x	1.2x	28.3x
kron-logn17 (U)	131.1	10229	$29937/78/378$	6	4447	1.7x	0.8x	16.7x
kron-logn18(U)	262	21166	$49164/81/454$	6	5292	1.3x	0.9x	30.0x
kron-logn19(U)	524	43563	$80676/83/541$	6	5808	1.1x	0.9x	17.3x
kron-logn20(U)	1049	89241	$131505/85/641$	6	3984	1.1x	1.1x	24.6x
kron-logn21(U)	2097	182084	$213906/87/756$	6	2642	0.9x	1.2x	15.4x
soc-Epinio1(D)	76	509	$1801/7/26$	11	848	8.0x	2.3x	16.7x
Linux-call(D)	324	1209	$712/4/6$	45	432	6.5x	1.2x	3.2x
web-Stanf (D)	282	2313	$255/8/11$	147	160	6.8x	0.8x	3.3x
com-LiveJ (D)	3998	69362	$14815/17/43$	14	1286	0.9x	0.8x	9.6x
com-Orkut(U)	3072	234370	$33133/76/155$	8	1698	0.9x	0.8x	6.4x
soc-LiveJour1(D)	4848	68994	$20293/14/36$	15	940	1.3x	0.9x	7.5x
mawi-12345(U)	18571	38040	$16.4 \times 10^6/2/3806$	11	1330	5.7x	0.7x	28.4x
mawi-20000(U)	35991	74485	$32.5 \times 10^6/2/5414$	11	1357	5.2x	0.7x	29.1x

TABLE III: Parameters, experimental MTEPs and speedup over the SuiteSparse:GraphBLAS ((**GBLAS**)**x**) libraries, obtained with the TurboBFS algorithms for a set of big graphs.

File	$V \times 10^6$	$E \times 10^6$	degree($\max/\mu/\sigma$)	d	MTEPs	(GBLAST)x	(gunrock)x	(GBLAS)x
kmer-P1a(U)	139	298	$40/2/1$	487	110	OOM	OOM	10.0x
kmer-A2a(U)	171	361	$40/2/1$	515	109	OOM	OOM	11.5x
kmer-V1r(U)	214	465	$8/2/1$	342	240	OOM	OOM	19.1x
it-2004(D)	41	1151	$9964/28/67$	50	892	OOM	OOM	3.3x
twitter7(D)	42	1468	$3 \times 10^6/35/2420$	13	1537	OOM	OOM	10.2x
GAP-twitter(D)	62	1468	$3 \times 10^6/24/1990$	15	811	OOM	OOM	8.3x
GAP-web(D)	51	1930	$12869/38/78$	56	1038	OOM	OOM	4.0x
sk-2005(D)	51	1949	$12870/39/78$	54	1147	OOM	OOM	4.6x

values for the depth (d) of the BFS tree.

V. Related work

The Breadth-First Search (BFS) algorithm is one of the most important building blocks for more sophisticated graph algorithms with a wide range of applications for modeling human brain [4], omics data [11], or social networks [12]. For example, the BFS algorithm is the first stage of the betweenness centrality (BC) algorithm due to Brandes [7]. Due to its importance and complexity, the Graph 500 benchmark uses BFS as one of the algorithms for ranking supercomputers [2].

The BFS algorithm had been used to evaluate the performance of practically all the high performance graphs processing libraries such as Ligra for shared memory machines [18], Gunrock on the GPU [19], SuiteSparse:GraphBLAS [1], the implementation fo the GraphBLAS standard, and the GPU based GraphBLAST library built over the GrapBLAS library [20].

As far as we know, the first BFS algorithm in the language of linear algebra was described on chapter 6 of reference [13] as a first step of the Brandes' betweenness centrality algorithm [7], [13]. This algorithm was implemented on the SuiteSparse:GraphBLAS and GraphBLAST libraries. In reference [20], the performances of both GPU-Based BFS algorithms were compared.

GraphBLAST and gunrock algorithms use a combined top-down bottom-up BFS algorithms that requires to store the arrays of the CSC and CSR formats simultaneously on the GPU for directed graphs, increasing the space complexity of the algorithms and limiting the size of the graphs for which the BFS can be computed. Our approach for designing and implementing the algorithms in TurboBFS differed from the GraphBLAST and the gunrock approaches, because we used highly scalable algorithms which were simpler and hence with less overhead. We also reduced the memory footprint of the TurboBFS algorithms by using only the CSC format for both directed and undirected graphs, 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, allowing us to compute the BFS for graphs with higher number of edges than those computed by GraphBLAST and the gunrock libraries on the same GPU. This reduction in space complexity also increased the performance of the TurboBFS algorithms.

VI. Summary and Future Work

In this paper, we designed and implemented TurboBFS, a highly scalable GPU-based set of BFS algorithms in the language of linear algebra. The algorithms in TurboBFS are applicable to unweighted, directed and undirected graphs represented by sparse adjacency matrices. The design goals of TurboBFS 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 algorithms. Our experiments showed that the algorithms in TurboBFS obtained up to 40 GTEPs, and were on average 15.7x, 5.8x, and 1.8x faster than the state-of-the-art algorithms implemented on the SuiteSparse:GraphBLAS, GraphBLAST, and gunrock libraries respectively. Our algorithms were able to compute the BFS for relatively big graphs for which the GraphBLAST and gunrock libraries ran out of memory, asserting our strategy of reducing the GPU global memory footprint required by our algorithms.

Our future work will be focused on improving the performance of the algorithms in TurboBFS, especially the performance of the algorithms computing the vector sparse matrix multiplication operations. Our goal will be to design and implement GPU-based BFS algorithms with higher performance than the state-of-the-art algorithms available on the GraphBLAST and gunrock libraries. We are also working on implementing a high-performance, highly scalable linear algebra version of the vertex and edge betweenness centrality algorithms proposed by Brandes [7], [13], using the GPU-based BFS algorithms presented in this paper.

VII. Acknowledgements

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] M. Aznaveh et al., "Parallel GraphBLAS with OpenMP", *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*, 2020, pp. 138-148

[2] D.A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, W. Mann and Theresa Meuse, "HPCS Scalable Synthetic Compact Applications 2 Graph Analysis (SSCA 2 v2.2 Specification)", 5 September 2007.http://graph500.org

[3] N.Bell and M. Garland. "Efficient Sparse Matrix-Vector Multiplication in CUDA", NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[4] D. S Bassett and Ol. Sporns, "Network neuroscience", Nat Neurosci. 2017 February 23; 20(3): 353–364

[5] S. Beamer, K. Asanovic and D. Patterson, "Direction-optimizing Breadth-First Search," SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, 2012, pp. 1-10.

[6] S. Beamer, A. Buluç, K. Asanovic and D. Patterson, "Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search," 2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, Cambridge, MA, 2013, pp. 1618-1627.

[7] U. Brandes. "On variants of shortest-path betweenness centrality and their generic computation", *Social Networks*, **30**, No. 2, 2008, pp.136-145.

[8] J. Cheng, M. Grossman and T. McKercher, *Professional CUDA C Programming*, John Wiley and Sons, Ltd., Indianapolis, Indiana, USA, 2014.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, Cambridge, USA, 2009.

[10] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection", *ACM Transactions on Mathematical Software* **38**, 1, Article 1, December 2011, https://sparse.tamu.edu.

[11] N. Gehlenborg, S. O'Donoghue, N. Baliga, et al. "Visualization of omics data for systems biology", Nat Methods 7, S56–S68 (2010)

[12] W. Jiang, G.Wang, Z. A. Bhuiyan, Ji. Wu, "Understanding Graph-based Trust Evaluation in Online Social Networks: Methodologies and Challenges", ACM Computing Surveys, Volume 49 , Issue 1, July 2016, Article No.: 10, pp 1–35

[13] J. Kepner and J. Gilbert (editors), *Graph Algorithms in the Language of Linear Algebra*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, USA, 2011.

[14] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T.A. Davis, M. Henderson, Y. Hu and R. Sandstrom,"The SuiteSparse Matrix Collection Website Interface," Journal of Open Source Software 4, 35 (March 2019), pages 1244-1248, https://sparse.tamu.edu.

[15] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection", http://snap.stanford.edu/data, jun 2014.

[16] NVIDIA corporation, *CUDA C++ PROGRAMMING GUIDE*, PG-02829-001 V11.1, June 2020

[17] Y. Pan, Y. Wang, Y. Wu, C. Yang and J. D. Owens, "Multi-GPU Graph Analytics," 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, 2017, pp. 479-490.

[18] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory", PPoPP '13: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, February 2013, pp.135–146

[19] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel and J. D. Owens, "Gunrock: A High-Performance Graph Processing Library on the GPU". PPoPP '16: 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming Barcelona, Spain, March 2016, pp. 1–12.

[20] C. Yang, A. Buluc and J. D. Owens, "GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU", arXiv:1908.01407v3 [cs.DC]