LAGraph: A Graph Algorithm and Network Analysis Library for GraphBLAS

Timothy A. Davis*, Timothy G. Mattson†, James Kitchen‡, Scott McMillan§,
Gábor Szárnyas¶, Erik Welch‡, David A. Bader‖

*Texas A&M University
†Parallel Computing Labs, Intel Corporation, Ocean Park, WA
‡Anaconda, Inc.
§ Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA
¶CWI Amsterdam, The Netherlands
‖New Jersey Institute of Technology

***Abstract*—design decisions, easy mode/expert mode, GAP algorithms, ...**

***Index Terms*—Graph Processing, Graph Algorithms, Graph Analytics, Linear Algebra, GraphBLAS**

I. Introduction

LAGraph is a library of Graph Algorithms based on the GraphBLAS

Key contributions:

- document design decisions for LAGraph
- present a concise notation for GraphBLAS algorihms
- algorithms of the GAP benchmark suite [4] used in the IISWC benchmark paper [2]
- improve data ingestion performance, e.g. using SIMD techniques [13]

Recently, numerous graph-specific have targeted GPUs such as the GraphBLAS Template Library (GBTL) [21], Gunrock [18] and GraphBLAST [20], and FPGAs [5].

However, in the near future we expect even more heterogeneous hardware architectures including graph-specific hardware based on the Programmable Integrated Unified Memory Architecture (PIUMA) [1]. Additionally, graph processing workloads can be offloaded to machine learning accelerators, e.g., Tensor Processing Units (TPUs) [11], systolic arrays using reconfigurable dataflow architecture [9], sparse linear algebra-based deep learning accelerators [14].

Previous GraphBLAS design papers: theory [15], C API [17], C++ API [7], distributed API [6], LAGraph [16]

```
int main() {
    return 0; // return zero
}
```

Listing 1: Example

[1]

Jim

II. Design Decisions

[1]A non peer-reviewed comparison of 6 popular graph algorithms libraries is available at https://www.timlrx.com/blog/benchmark-of-popular-graph-network-packages-v2.

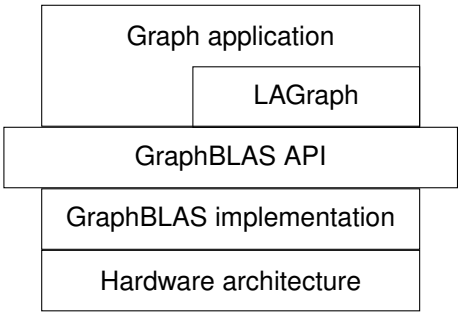

Fig. 1: Separation of concerns using the GraphBLAS API.

We investigate the following design questions:

- data structure for representing a graph
- basic/advanced mode
- algorithm calling conventions
- error handling
- user-contributed algorithms
- ———
- how to work around typecasting being expensive in GraphBLAS
- terminology: properties (parameters? features?)
- multi-threadable

A. Core data structure

The primary data structure in LAGraph is the `LAGraph_Graph` which consists of primary components and cached properties. The data structure is not opaque, providing the user with full access to access and modify all internal components, unlike the way GraphBLAS objects behave.

The primary components are a GraphBLAS matrix named `A` and an enumeration `kind`. The kind indicates how the matrix should be interpreted. Currently, the only kinds defined are `LAGRAPH_ADJACENCY_UNDIRECTED` and `LAGRAPH_ADJACENCY_DIRECTED`, but more options will be added in the future.

Cached properties include the transpose of A, the row degrees, column degrees, etc. They can be computed from the primary components, but doing so repeatedly for each algorithm utilizing A would be wasteful. Having them live inside the Graph object helps simplify algorithm call signatures. Utility functions exist to compute each cached property.

The following will compute the transpose of G->A and store it as G->AT.

```
LAGraph_Property_AT(G, msg);
```

Listing 2: Utility to compute the transpose

Following this call, any algorithm which is given G will have access to both A and its transpose.

Because the Graph object is opaque, any piece of code may set the transpose as well. For instance, if an algorithm computes the transpose as part of its normal logic, it could directly set G->AT. The expectation is that the Graph object will always remain consistent. If G->A is modified, all cached properties must be either be set as unknown or modified to reflect the change. Properties which are not known are set to NULL or LAGRAPH_BOOLEAN_UNKNOWN in the case of Boolean properties. This expectation is a convention that all LAGraph algorithm implementers are expected to follow.

Creating the Graph object is performed by

```
LAGraph_Graph G;
LAGraph_New(&G, &M, LAGRAPH_DIRECTED_ADJACENCY);
```

Listing 3: Constructing an LAGraph_Graph object

Following this call, M will be NULL. The matrix previously pointed to by M now lives at G->A. This "move" constructor helps avoid memory freeing errors.

B. User modes

Algorithms in LAGraph target two types of user modes: Basic and Advanced. The Basic user mode is for those wanting things to "just work", are less concerned about performance, and may be less experienced with the library. The Advanced user mode is for those whose primary concern is performance and are willing to conform to stricter requirements to achieve that goal.

Algorithms targeting the Basic mode typically have limited options. Often, there will only be one function for a given algorithm. Under the hood, that single algorithm might take different paths depending on the shape or size of the input graph. The idea is that a basic user wants to compute Pagerank or Betweenness Centrality, but doesn't want to have to understand the five different ways to compute them. They simply want the correct answer.

Algorithms targeting the Advanced mode are often highly specialized implementations of an algorithm. The Advanced mode user is expected to understand details such as push-pull and batch mode and why different techniques are better for each graph. Advanced mode algorithms are very strict in their input. If the input doesn't match the expected kind, an error will be raised.

Advanced mode algorithms will also raise an error if a cached property is needed by an algorithm, but is not currently available on the Graph object. While Basic mode algorithms are free to compute and cache properties on the Graph object, Advanced mode algorithms never will. The idea is to never surprise the user with unexpected additional computation. An Advanced mode user must opt-in to all computations.

Often, Basic mode algorithms will inspect the input, possibly compute properties or transform the data, and finally call one of the Advanced mode algorithms to do the actual work on the graph. Having these two user modes allows LAGraph to target a wider range of users who vary in their experience with graph algorithms.

C. Algorithm calling conventions

Algorithms in LAGraph follow a general calling convention.

```
int algorithm
(
    // outputs:
    TYPE *out1,
    TYPE *out2,
    ...
    // input/output
    TYPE inout,
    ...
    // inputs
    TYPE input1,
    TYPE input2,
    ...
    // error message holder
    char *msg
)
```

Listing 4: Calling convention in LAGraph

The return value is always an int with the following meaning:

- =0 -> success
- <0 -> error
- >0 -> warning

The meaning of a given error or warning value is algorithm-specific and should be listed in the documentation for the algorithm.

Outputs appear first and are passed by reference. A pointer should be created by the caller, but memory will be allocated by the algorithm. If the output is not needed, a NULL is passed and the algorithm will not return that output.

Input/Output arguments are passed by value. The expectation is that the object will be modified. This supports things like batch mode in which a frontier is updated and returned to the caller. It also supports Basic mode algorithms which may modify a Graph object by adding cached properties.

Inputs are passed by value and should never be changed by the algorithm.

The final argument of any LAGraph algorithm is the error message holder. This must be char[] of size LAGRAPH_MSG_LEN. When the algorithm returns an error or a warning, a message may be placed in this array as additional information. Because the caller creates this array, the caller must free the memory or reuse it as appropriate. If the algorithm is successful, it should fill the message array with an empty string to clear any previous message.

D. Error handling

Because every algorithm in LAGraph can return an error, the return value of every call should be checked before proceeding. To make this less burdensome for a C-based library, LAGraph provides a convenience macro which works similar to try/catch in other languages.

```
#define LAGraph_TRY(LAGraph_method)
{
    int LAGraph_status = LAGraph_method;
    if (LAGraph_status < 0)
    {
        LAGraph_CATCH (LAGraph_status);
    }
}
```

Listing 5: LAGraph Try/Catch Utility Macro

`LAGraph_CATCH` can be defined before an algorithm and will be called in the event of an error. This allows for proper freeing of memory and other necessary tasks.

A similar macro, `GrB_TRY`, will call `GrB_CATCH` when making GraphBLAS calls which return a `GrB_Info` value other than `GrB_SUCCESS` or `GrB_NO_VALUE`.

`LAGraph_TRY` and `GrB_TRY` provide an easy to use and easy to read method for dealing with error checking while writing graph algorithms.

E. Contributing algorithms

The LAGraph project welcomes contributions from all graph practitioners who understand the GraphBLAS vision of using the language of linear algebra to express graph computations. However, as a matter of practical concern, many users want a stable experience when using LAGraph for doing real work. To balance these, the LAGraph repository will have both a stable and an experimental folder.

New algorithms or modifications of existing algorithms will first be added to the experimental folder. The release schedule of experimental algorithms will generally be much faster than the stable release and there is no expectation of a bug-free experience. The goal is to generate lots of ideas and allow uninhibited contributions to push the boundary of what is possible with the GraphBLAS. The stable release will be fully tested and will move much slower, targeting the needs of those who want to use LAGraph as a complete, production-grade library rather than as a research project.

III. Notation

IV. Algorithms

GAP algorithms: BFS, SSSP, TC, BC, PR. Not sure whether CC should be included.

A. BFS

push/pull [19]

GraphBLAST stores both directions and picks automatically [20]

B. Betweenness Centrality

also does push/pull

C. PageRank

D. SSSP

E. Triangle Count

(Not too interesting from an API design point of view.)

F. Connected Components

Needs a few GxB extensions.

V. Evaluation

Tim D

A. SuiteSparse Extensions

In a prior paper ([2]), an early draft of SuiteSparse:GraphBLAS v4.0.0 (Aug 2020) was compared with the GAP benchmark [3] and four other graph libraries. This prior version of SS:GrB included two primary data structures for its sparse matrices: compressed sparse vector, and a hypersparse variant [8], both held by row or by column. It included a draft implementation of a bitmap data structure that could only be used in a prototype breadth-first search. Since then, SuiteSparse:GraphBLAS v4.0.3 has been released, with full support for bitmap and full matrices for all its operations. In an m-by-n bitmap matrix, the values are held in a full array of size mn, and another `int8_t` array of size mn holds the sparsity pattern of the matrix. A full matrix is a simple dense array of size mn.

The bitmap format is particularly important for the "pull" phase of an algorithm, as used in direction-optimizing breadth-first-search [3]. The GAP benchmark suite uses this method by holding its frontier as a bitmap in the pull step and as a list in the push step. The GAP BFS was shown to be typically the fastest BFS amongst the 6 graph libraries compared in [2] (for 4 of the 5 benchmark graphs). With the addition of the bitmap format to SS:GrB, LAGraph+SS:GrB is able to come within a factor of 2 or so of the performance of the highly-tuned BFS GAP benchmark (see the results in the next section), for those 4 graphs. At the same time, however, the BFS is very easily expressed in LAGraph as an easy-to-read and easy-to-write code. This enables non-experts to obtain a reasonably high level of performance with modest programming effort when writing their own graph algorithms.

Direction-optimization is incredibly simple to add to an LAGraph algorithm. For example, a batched direction-optimizing betweenness-centrality (BC) algorithm in LAGraph only requires a simple heuristic to determine which direction to use, followed by masked matrix-matrix multiplication with the matrix or its transpose: $F\langle\neg P\rangle = FB'$ (the pull) or $F\langle\neg P\rangle = FA$ (the push), where A is the adjacency matrix of the graph and $B = A'$ is its explicit transpose, F is the frontier, and the complemented mask $\neg P$ is the set of unvisited nodes. The multiplication FB' relies on the descriptor to represent the transpose of B, which is not explicitly transposed. In the backward phase, the pull step is $W = WA'$ while the push is $W = WB$, where W is the 4-by-n matrix in which centrality is accumulated.

Additional optimizations added to SS:GrB in the past year include a *lazy sort*. Normally, SS:GrB keeps its vectors sorted

(row vectors in a CSR matrix, or column vectors if the matrix is held by column), with entries sorted in ascending order of column or row index, respectively. This simplifies the many algorithms that operate on a `GrB_Matrix`. However, some algorithms naturally produce a jumbled result (matrix multiply in particular), while many algorithms are tolerant of jumbled input matrices. We thus allow the sort to be left pending. The lazy sort joins two other kinds of pending work in SS:GrB: *pending tuples*, and *zombies*. A pending tuple is an entry that is held inside a matrix in an unsorted list, awaiting insertion into the CSR/CSC format of a `GrB_Matrix`. A zombie is the opposite: it is an entry in the CSR/CSC format that has been marked for deletion, but has not yet been deleted from the matrix. With the lazy sort, the sort is postponed until another algorithm requires sorted input matrices. If the sort is lazy enough, it might never occur, which is the case for the LAGraph BFS.

Another useful addition to SS:GrB is the new positional binary operators. The BFS relies on the ANY-SECONDI semiring to compute a single step, $q\langle\neg\pi\rangle = q'A$, where q is the current frontier π is the parent vector, and A is the adjacency matrix.

Consider a matrix multiply for conventional linear algebra, where the PLUS monoid sums a set of t entries to obtain a single scalar for computing $c_{ij} = \sum a_{ik}b_{kj}$ in the matrix multiply $C = AB$. The ANY monoid performs the reduction of t entries to a single number by merely selecting any one of the t entries as the result c_{ij}. The selection is done non-deterministically, allowing for a benign race condition. In the BFS, this corresponds to selecting any valid parent of a newly discovered node. Indeed, the creation of the ANY operator was inspired by Scott Beamer's `bfs.cc` method in the GAP benchmark, which has the same benign race condition. The ANY monoid translates the concept of this benign race condition to construct a valid BFS tree into a linear algebraic operation, suitable for implementation in GraphBLAS.

The SECONDI operator is the multiplicative operator in the ANY-SECONDI semiring, where the result of $a_{ik}b_{kj}$ is simply the index k in the semiring for $C = AB$. This gives the id of the parent node for a newly discovered node in the next frontier. The ANY monoid then selects any valid parent k.

B. Performance Results

Our benchmark environment is an NVIDIA DGX Station (donated to Texas A&M by NVIDIA in support of this research). It includes a 20-core Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz, with 40 threads. All codes were compiled with gcc 5.4.0 (-O3). All default settings were used, which means that hyperthreading was enabled. The system has 256GB of RAM in a single socket (no NUMA effects). LAGraph (Jan 29, 2021) and SuiteSparse:GraphBLAS 4.0.3 (Jan 19, 2021) were used. The NVIDIA DGX Station includes four P100 GPUs, but no GPUs were used by this experiment (a GPU-accelerated SS:GrB is in progress). Table I lists the run time (in seconds) for the GAP benchmark and LAGraph+SS:GrB for the 6 algorithms on the 5 benchmark matrices. The benchmark matrices are listed in Table II.

Algorithm : package	graph, with run time in seconds				
	Kron	Urand	Twitter	Web	Road
BC : GAP	31.52	46.36	10.82	3.01	1.50
BC : SS	26.85	31.78	10.04	9.25	51.91
BFS : GAP	.31	.58	.22	.34	.25
BFS : SS	.52	1.31	.33	.67	3.33
PR : GAP	19.81	25.29	15.16	5.13	1.01
PR : SS	21.96	27.75	17.22	9.30	1.34
CC : GAP	.53	1.66	.23	.22	.05
CC : SS	3.42	4.59	1.48	1.97	1.00
SSSP : GAP	4.91	7.23	2.02	.81	.21
SSSP : SS	17.62	25.62	8.44	9.67	48.49
TC : GAP	374.08	21.83	79.58	22.18	.03
TC : SS	943.47	34.10	242.36	35.15	.29

TABLE I: Run time of GAP and LAGraph+SS:GrB

graph	nodes	entries in A	graph kind
Kron	134,217,726	4,223,264,644	undirected
Urand	134,217,728	4,294,966,740	undirected
Twitter	61,578,415	1,468,364,884	directed
Web	50,636,151	1,930,292,948	directed
Road	23,947,347	57,708,624	directed

TABLE II: Benchmark matrices (https://sparse.tamu.edu/GAP)

With the simple addition of the bitmap (needed for the pull step), the push/pull optimization in BC resulted in a nearly 2x performance gain in the GraphBLAS method for the largest matrices, as compared to the SS:GrB version used for the results presented in [2].

With this change the BC method in LAGraph+SS:GrB is not only expressible in a simple, elegant code, but it is also faster than the highly-tuned GAP benchmark method, `bc.cc`, for the three largest matrices (1.2x for Kron, 1.5x for Urand, and 1.08x for Twitter).

We expect the BC in LAGraph+SS:GrB to become faster still in the next release because we have not yet fully exploited the lazy sort. The frontier matrix F is left jumbled by the lazy sort, but it is sorted right away by a subsequent assignment. Most uses of `GrB_assign` are intolerant of jumbled input matrices, so it sorts them on input. However, `GrB_assign` includes about 40 internal variations, a few of which do not actually require sorted input matrices. The particular method used in `GrB_assign` in the LAGraph BC method is one of those methods, so this would be simple to exploit. With this change, the sort would be so lazy that it would *never* occur. The frontier would be computed, left jumbled, used in subsequent computations, and then recomputed (and thus discarded), just as we currently do in the LAGraph BFS.

With the addition of bitmap format (which makes push/pull optimization very simple to express, and very fast) and the ANY-SECONDI semiring, the BFS of a directed or undirected graph is easily expressed in GraphBLAS, and has a performance that is only about 2x slower than the GAP benchmark. We expect the remaining 2x performance gap arises from two issues:

1) The GAP assumes that the graph has fewer than 2^{32}

nodes and edges, and thus uses 32-bit integers throughout. GraphBLAS is written for larger problems, and thus relies solely on 64-bit integers. This cannot be easily changed in GraphBLAS, but rather than "fixing" GraphBLAS to use smaller integers, it is the GAP algorithms that would need to be updated for larger graphs in the future. In the current GAP benchmark graphs, two graphs are chosen with almost exactly 4 billion edges. Graphs of current interest in large data science can easily exceeed 2^{32} nodes and edges [12].

2) In GraphBLAS, the BFS must be expressed as two calls. The first computes $q\langle\neg\pi\rangle = q'A$, and the second updates the parent vector, $\pi\langle q\rangle = q$:

```
GrB_vxm (q, pi, NULL, semiring, q, A, GrB_DESC_RSC) ;
GrB_assign (pi, q, NULL, q, GrB_ALL, n, GrB_DESC_S) ;
```

In Scott Beamer's `bfs.cc`, these two steps are fused, and the matrix-vector multiplication can write its result directly into the vector `pi`. This could be implemented in a future GraphBLAS library, since the GraphBLAS API allows for a non-blocking mode where work is queued and done later, thus enabling a fusion of these two steps. SS:GrB exploits the non-blocking mode (for its lazy sort, pending tuples, and zombies) but does not *yet* exploit the fusion of `GrB_vxm` and `GrB_assign`. We intend to exploit this in the future.

Note that LAGraph+SS:GrB is quite slow for many algorithms (all but PR) on the Road graph. The primary reason for this is the high diameter of the Road graph (about 6000). This requires 6000 iterations of GraphBLAS in the BFS, each with a tiny amount of work. Each call to GraphBLAS does several `malloc` and `free`s, and in some cases the workspace must be initialized. A future version of SS:GrB is planned that will eliminate this work entirely, by implementing an internal memory pool. There may be other overheads, but we hope that a memory pool, fusion to fully exploit non-blocking mode, and other optimization will address this large performance gap for the Road graph for these algorithms.

LAGraph+SS:GrB is also up to 3x slower than the GAP for the triangle counting problem (for all but the Road graph, where it is even slower). This performance gap can be eliminated entirely in the future, if the `GrB_mxm` and `GrB_reduce` are combined in a single fused step, by a full exploitation of the GraphBLAS non-blocking mode. The current method computes $C\langle L\rangle = LU'$, followed by the reduction of C to a single scalar. The matrix C is then discarded. All that GraphBLAS needs is a fused kernel that does not explicitly instantiate the temporary matrix C.

VI. Utility Fuctions

sort, diag/invdiag (?), equal

VII. Conclusion

Future work - ideas:

- Create a Python wrapper for LAGraph
- Implement the LDBC Graphalytics benchmark [10]
- Improve data ingestion performance using e.g., SIMD instructions [13]

Acknowledgements

add acks

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM21-0xxx].

G. Szárnyas was partially supported by the SQIREL-GRAPHS NWO project.

References

[1] S. Aananthakrishnan *et al.*, "PIUMA: Programmable Integrated Unified Memory Architecture," *CoRR*, vol. abs/2010.06277, 2020. [Online]. Available: https://arxiv.org/abs/2010.06277

[2] A. Azad *et al.*, "Evaluation of graph analytics frameworks using the GAP Benchmark Suite," in *IEEE*. IEEE, 2020, pp. 216–227. [Online]. Available: https://doi.org/10.1109/IISWC50251.2020.00029

[3] S. Beamer *et al.*, "Direction-optimizing breadth-first search," in *SC*. IEEE/ACM, 2012. [Online]. Available: https://doi.org/10.1109/SC.2012.50

[4] ——, "The GAP Benchmark Suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[5] M. Besta, D. Stanojevic, J. de Fine Licht, T. Ben-Nun, and T. Hoefler, "Graph processing on FPGAs: Taxonomy, survey, challenges," *CoRR*, vol. abs/1903.06697, 2019. [Online]. Available: http://arxiv.org/abs/1903.06697

[6] B. Brock *et al.*, "Considerations for a distributed GraphBLAS API," in *GrAPL at IPDPS*. IEEE, 2020, pp. 215–218. [Online]. Available: https://doi.org/10.1109/IPDPSW50202.2020.00048

[7] ——, "A roadmap for the GraphBLAS C++ API," in *GrAPL at IPDPS*. IEEE, 2020, pp. 219–222. [Online]. Available: https://doi.org/10.1109/IPDPSW50202.2020.00049

[8] A. Buluç and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *IPDPS*. IEEE, 2008, pp. 1–11. [Online]. Available: https://doi.org/10.1109/IPDPS.2008.4536313

[9] G. F. Grohoski, S. J. Luttrell, R. Prabhakar, R. Sivaramakrishnan, and M. K. Shah, "Virtualization of a reconfigurable data processor," U.S. Patent 20 200 257 643A1, Aug. 13, 2020.

[10] A. Iosup *et al.*, "LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms," *VLDB*, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p1317-iosup.pdf

[11] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*. ACM, 2017. [Online]. Available: https://doi.org/10.1145/3079856.3080246

[12] J. Kepner, C. Meiners, C. Byun, S. McGuire, T. Davis, W. Arcand, J. Bernays, D. Bestor, W. Bergeron, V. Gadepally, R. Harnasch, M. Hubbell, M. Houle, M. Jones, A. Kirby, A. Klein, L. Milechin, J. Mullen, A. Prout, A. Reuther, A. Rosa, S. Samsi, D. Stetson, A. Tse, C. Yee, and P. Michaleas, "Multi-temporal analysis and scaling relations of 100,000,000,000 network packets," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–6.

[13] G. Langdale and D. Lemire, "Parsing gigabytes of JSON per second," *VLDB J.*, vol. 28, no. 6, pp. 941–960, 2019. [Online]. Available: https://doi.org/10.1007/s00778-019-00578-5

[14] S. Lie, G. R. Lauterbach, M. E. James, M. Morrison, and S. Arekapudi, "Dataflow triggered tasks for accelerated deep learning," U.S. Patent 10 614 357B2, Apr. 7, 2020.

[15] T. Mattson *et al.*, "Standards for graph algorithm primitives," in *HPEC*. IEEE, 2013. [Online]. Available: https://doi.org/10.1109/HPEC.2013.6670338

[16] ——, "LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS," in *GrAPL at IPDPS*, 2019. [Online]. Available: https://doi.org/10.1109/IPDPSW.2019.00053

[17] T. G. Mattson *et al.*, "GraphBLAS C API: Ideas for future versions of the specification," in *HPEC*. IEEE, 2017. [Online]. Available: https://doi.org/10.1109/HPEC.2017.8091095

[18] Y. Wang *et al.*, "Gunrock: GPU graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, pp. 3:1–3:49, 2017. [Online]. Available: https://doi.org/10.1145/3108140

[19] C. Yang *et al.*, "Implementing push-pull efficiently in GraphBLAS," in *ICPP*. ACM, 2018, pp. 89:1–89:11. [Online]. Available: https://doi.org/10.1145/3225058.3225122

[20] ——, "GraphBLAST: A high-performance linear algebra-based graph framework on the GPU," *CoRR*, vol. abs/1908.01407, 2019. [Online]. Available: http://arxiv.org/abs/1908.01407

[21] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan, "Gbtl-cuda: Graph algorithms and primitives for gpus," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 912–920.