LAGraph: A Graph Algorithm and Network Analysis Library for GraphBLAS

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

[*]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*—IMHO, the motivation for this work can be summarized as follows (Gabor):**

To ensure portability, the primitives of GraphBLAS are quite low-level.

Formulating efficient algorithms using these primitives requires expertise from the user.

To mitigate this challenge, the LAGraph library provides efficient implementations through a convenient API that can be used by users. Additionally, it demonstrates best practices for algorithm implementers.

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 [5] used in the IISWC benchmark paper [3]
- improve data ingestion performance, e.g. using SIMD techniques [18]

Recently, numerous graph processing systems have targeted GPUs such as the GraphBLAS Template Library (GBTL) [29], Gunrock [25] and GraphBLAST [28], as well as FPGAs [6].

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) [2]. Additionally, graph processing workloads can be offloaded to machine learning accelerators, e.g., Tensor Processing Units (TPUs) [16], systolic arrays using reconfigurable dataflow architecture [13], sparse linear algebra-based deep learning accelerators [20].

Previous GraphBLAS design papers: theory [21], C API [23], C++ API [8], distributed API [7], LAGraph [22]

Popular libraries: igraph [11], NetworkX [1], SNAP [19] offer very limited or no parallelism.

TODO: add two paragraphs with a high-level overview of LAGraph

```
// This is a minted code block inside a float region
int main() {
    return 0; // return zero
}
```

Listing 1: Example 2

1

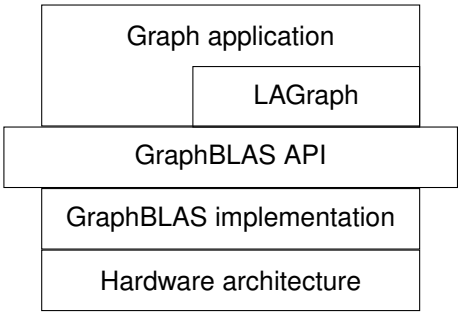

Fig. 1: Separation of concerns using the GraphBLAS API.

II. Design Decisions

TODO: Jim

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

[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.

```
1  typedef struct LAGraph_Graph_struct
2  {
3      GrB_Matrix    A;      // adjacency matrix of the graph
4      LAGraph_Kind kind;    // kind of graph: directed, etc.
5
6      // cached properties
7      GrB_Matrix    AT;     // transpose of A
8      GrB_Vector    row_degree;
9      GrB_Vector    col_degree;
10     LAGraph_BooleanProperty A_pattern_is_symmetric;
11     int64_t       ndiag;  // -1 if unknown
12 } *LAGraph_Graph;
13
14 typedef struct LAGraph_Graph_struct *LAGraph_Graph;
15
16 // creating a graph
17 GrB_Matrix M;
18 // ...construction of M omitted
19
20 LAGraph_Graph G;
21 LAGraph_New(&G, &M, LAGRAPH_DIRECTED_ADJACENCY);
22
23 // operating on properties
24 LAGraph_Property_AT(G, msg);  // compute/cache
```

Listing 2: `LAGraph_Graph` data structure and methods.

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. This data structure is shown at the top of Listing 2 and defined ultimately on Line 14.

The primary components of this struct 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. Creating the Graph object is performed on Line 21 of Listing 2. 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.

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. For example, Line 24 of Listing 2 will compute the transpose of `G->A` and store it as `G->AT`. 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.

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
)
```

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);
    }
}
```

`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

"guiding principles": understandable (biggest probability of first guess to be correct), similar to existing notations [9]

TODO: initial draft by Gabor

TODO: explain masks

TODO: explain replace/merge as per Scott's email

TODO: revise Scott, Tim D, and Tim M

TODO: consider adding semiring table

what do masks mean?

the mask restricts the scope of the computation to the elements selected by the mask ($\mathbf{m}$) or the complement of these elements ($\neg\mathbf{m}$)

there are variations based on (1) how the elements are selected (based on the *values* in the mask or the *structure* of the mask) (2) how the elements outside the selected ones are treated (they are *replaced* with implicit zeros or they are kept intact and *merged* with the results of the computation)

by default, masks should use *merge* semantics, i.e. the computation can only effect elements selected by the mask, elements outside the mask are unaffected $\mathbf{w}\langle\mathbf{m}\rangle$ and $\mathbf{w}\langle\neg\mathbf{m}\rangle$

replace semantics, i.e. they annihilate all elements outside the mask. this is denoted with $\mathbf{w}\langle\mathbf{m},\mathrm{r}\rangle$ and $\mathbf{w}\langle\neg\mathbf{m},\mathrm{r}\rangle$

by default, element values in the mask are checked and elements with zero value are not used to avoid this, we use *structural masks* $\mathbf{w}\langle s(\mathbf{m})\rangle$ and $\mathbf{w}\langle\neg s(\mathbf{m})\rangle$

Combining *replace semantics* and *structural masks* is possible: $\mathbf{w}\langle s(\mathbf{m}),\mathrm{r}\rangle$ $\mathbf{w}\langle\neg s(\mathbf{m}),\mathrm{r}\rangle$

Initializing scalars, vectors, and matrices (GraphBLAS methods):

- let: $s \in \mathbb{Q}_{64}$
- let: $\mathbf{w} \in \mathbb{Q}_{32}^{n}$
- let: $\mathbf{A} \in \mathbb{N}_{16}^{m\times n}$
- let: $\mathbf{A} \in \mathbb{Z}_{64}^{k\times m}$

a) Transposition: separate operator + descriptor ($^{\mathsf{T}}$)

vectors can be interpreted as row/column vectors, we do not transpose them manually

extensions

conversion table

operation/method	description	notation
`mxm`	matrix-matrix multiplication	$\mathbf{C}\langle\mathbf{M}\rangle\odot{=}\mathbf{A}\oplus.\otimes\mathbf{B}$
`vxm`	vector-matrix multiplication	$\mathbf{w}^\mathsf{T}\langle\mathbf{m}^\mathsf{T}\rangle\odot{=}\mathbf{u}^\mathsf{T}\oplus.\otimes\mathbf{A}$
`mxv`	matrix-vector multiplication	$\mathbf{w}\langle\mathbf{m}\rangle\odot{=}\mathbf{A}\oplus.\otimes\mathbf{u}$
`eWiseAdd`	element-wise addition using operator `op` on elements in the set union of structures of $\mathbf{A}/\mathbf{B}$ and $\mathbf{u}/\mathbf{v}$	$\mathbf{C}\langle\mathbf{M}\rangle\odot{=}\mathbf{A}\ \mathsf{op}_\cup\ \mathbf{B}$ $\mathbf{w}\langle\mathbf{m}\rangle\odot{=}\mathbf{u}\ \mathsf{op}_\cup\ \mathbf{v}$
`eWiseMult`	element-wise multiplication using operator `op` on elements in the set intersection of structures of $\mathbf{A}/\mathbf{B}$ and $\mathbf{u}/\mathbf{v}$	$\mathbf{C}\langle\mathbf{M}\rangle\odot{=}\mathbf{A}\ \mathsf{op}_\cap\ \mathbf{B}$ $\mathbf{w}\langle\mathbf{m}\rangle\odot{=}\mathbf{u}\ \mathsf{op}_\cap\ \mathbf{v}$
`extract`	extract submatrix from matrix $\mathbf{A}$ using indices $\boldsymbol{i}$ and indices $\boldsymbol{j}$ extract the ith row vector from matrix $\mathbf{A}$ extract the jth column vector from matrix $\mathbf{A}$ extract subvector from $\mathbf{u}$ using indices $\boldsymbol{i}$	$\mathbf{C}\langle\mathbf{M}\rangle\odot{=}\mathbf{A}(\boldsymbol{i},\boldsymbol{j})$ $\mathbf{w}\langle\mathbf{m}\rangle\odot{=}\mathbf{A}(i,:)$ $\mathbf{w}\langle\mathbf{m}\rangle\odot{=}\mathbf{A}(:,j)$ $\mathbf{w}\langle\mathbf{m}\rangle\odot{=}\mathbf{u}(\boldsymbol{i})$
`assign`	assign matrix to submatrix with mask for $\mathbf{C}$ assign scalar to submatrix with mask for $\mathbf{C}$ assign vector to subvector with mask for $\mathbf{w}$ assign scalar to subvector with mask for $\mathbf{w}$	$\mathbf{C}\langle\mathbf{M}\rangle(\boldsymbol{i},\boldsymbol{j})\odot{=}\mathbf{A}$ $\mathbf{C}\langle\mathbf{M}\rangle(\boldsymbol{i},\boldsymbol{j})\odot{=}s$ $\mathbf{w}\langle\mathbf{m}\rangle(\boldsymbol{i})\odot{=}\mathbf{u}$ $\mathbf{w}\langle\mathbf{m}\rangle(\boldsymbol{i})\odot{=}s$
`subassign (GxB)`	assign matrix to submatrix with submask for $\mathbf{C}(\boldsymbol{i},\boldsymbol{j})$ assign scalar to submatrix with submask for $\mathbf{C}(\boldsymbol{i},\boldsymbol{j})$ assign vector to subvector with submask for $\mathbf{w}(\boldsymbol{i})$ assign scalar to subvector with submask for $\mathbf{w}(\boldsymbol{i})$	$\mathbf{C}(\boldsymbol{i},\boldsymbol{j})\langle\mathbf{M}\rangle\odot{=}\mathbf{A}$ $\mathbf{C}(\boldsymbol{i},\boldsymbol{j})\langle\mathbf{M}\rangle\odot{=}s$ $\mathbf{w}(\boldsymbol{i})\langle\mathbf{m}\rangle\odot{=}\mathbf{u}$ $\mathbf{w}(\boldsymbol{i})\langle\mathbf{m}\rangle\odot{=}s$
`apply`	apply unary operator f with optional thunk k	$\mathbf{C}\langle\mathbf{M}\rangle\odot{=}f(\mathbf{A},k)$ $\mathbf{w}\langle\mathbf{m}\rangle\odot{=}f(\mathbf{u},k)$
`select`	apply select operator f with optional thunk k	$\mathbf{C}\langle\mathbf{M}\rangle\odot{=}\mathbf{A}\langle f(\mathbf{A},k)\rangle$ $\mathbf{w}\langle\mathbf{m}\rangle\odot{=}\mathbf{u}\langle f(\mathbf{u},k)\rangle$
`reduce`	row-wise reduce matrix to column vector reduce matrix to scalar reduce vector to scalar	$\mathbf{w}\langle\mathbf{m}\rangle\odot{=}[\oplus_j\,\mathbf{A}(:,j)]$ $s\odot{=}[\oplus_{i,j}\,\mathbf{A}(i,j)]$ $s\odot{=}[\oplus_i\,\mathbf{u}(i)]$
`transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle\odot{=}\mathbf{A}^\mathsf{T}$
`kronecker`	Kronecker multiplication using operator `op`	$\mathbf{C}\langle\mathbf{M}\rangle\odot{=}\mathrm{kron}(\mathbf{A},\mathsf{op},\mathbf{B})$
`dup`	duplicate matrix duplicate vector	$\mathbf{C}\leftarrowtail\mathbf{A}$ $\mathbf{w}\leftarrowtail\mathbf{u}$
`build`	matrix from tuples vector from tuples	$\mathbf{C}\leftarrowtail\{\boldsymbol{i},\boldsymbol{j},\boldsymbol{x}\}$ $\mathbf{w}\leftarrowtail\{\boldsymbol{i},\boldsymbol{x}\}$
`extractTuples`	extract index arrays ($\boldsymbol{i}$,$\boldsymbol{j}$) and value arrays ($\boldsymbol{x}$)	$\{\boldsymbol{i},\boldsymbol{j},\boldsymbol{x}\}\rightarrowtail\mathbf{A}$ $\{\boldsymbol{i},\boldsymbol{x}\}\rightarrowtail\mathbf{u}$
`extractElement`	extract element to scalar	$s=\mathbf{A}(i,j)$ $s=\mathbf{u}(i)$
`setElement`	set element	$\mathbf{C}(i,j)=s$ $\mathbf{w}(i)=s$

TABLE I: GraphBLAS operations and methods based on [12]. *Notation:* Matrices and vectors are typeset in bold, starting with uppercase ($\mathbf{A}$) and lowercase ($\mathbf{u}$) letters, respectively. Scalars including indices are lowercase italic (k, i, j) while arrays are lowercase bold italic ($\boldsymbol{x}$, $\boldsymbol{i}$, $\boldsymbol{j}$). $\oplus$ and $\otimes$ are the addition and multiplication operators forming a semiring and default to conventional arithmetic $+$ and $\times$ operators. $\odot$ is the accumulator operator. Operations can be modified via a descriptor; matrices can be transposed ($\mathbf{B}^\mathsf{T}$), the mask can be complemented, and the mask can be valued (shown above) or structural ($\mathbf{C}\langle s(\mathbf{M})\rangle$). A structural mask can also be complemented ($\mathbf{C}\langle\neg s(\mathbf{M})\rangle$). The result can be cleared after using it as input to the mask/accumulator step ($\mathbf{C}\langle\mathbf{M},\mathrm{r}\rangle$). Not all methods are listed (creating new ops, monoids, and semirings, clearing a matrix/vector, etc.). For the `assign` operation, the dimensions of the masks are the same as the output's: $\dim(\mathbf{M})=\dim(\mathbf{C})$ and $\dim(\mathbf{m})=\dim(\mathbf{w})$. For the `subassign` operation, the dimensions of the masks are the same as the submasks: $\mathrm{nrows}(\mathbf{M})=|\boldsymbol{i}|,\mathrm{ncols}(\mathbf{M})=|\boldsymbol{j}|$ for matrices and $\dim(\mathbf{m})=|i|$ for vectors.

IV. ALGORITHMS

A. Breadth-First Search (BFS)

The breadth-first search (BFS) builds on the observation that a vector-matrix multiplication $\mathbf{f}^\mathsf{T}\mathbf{A}$ expresses the navigation from the nodes selected by vector $\mathbf{f}$ in the graph represented by $\mathbf{A}$. The direction-optimizing push/pull BFS [4] is simple to express in GraphBLAS [27]. If $\mathbf{A}$ is held by row, then $\mathbf{f}^\mathsf{T}\mathbf{A}$ is a push step, while $\mathbf{Bf}$ is a pull step, where $\mathbf{B} = \mathbf{A}^\mathsf{T}$ is the explicit transpose of $\mathbf{A}$, also held by row. Other GraphBLAS libraries, e.g., GraphBLAST, store both directions and perform direction-optimization automatically [28]. The push-only BFS is shown in Alg. 1, while the push/pull BFS is Alg. 2.

The GraphBLAS BFS relies on the any.secondi semiring to compute a single step, $\mathbf{q}\langle\neg s(\mathbf{p})\rangle = \mathbf{q}^\mathsf{T}\mathbf{A}$, where $\mathbf{q}$ is the current frontier (using $\mathbf{q}$ as short for queue), $\mathbf{p}$ is the parent vector, and $\mathbf{A}$ is the adjacency matrix. This step assigns the parents of newly nodes, which do not yet have a parent, using the complemented structural mask $\langle\neg s(\mathbf{p})\rangle$.

Consider a matrix multiply for conventional linear algebra, where the $\oplus$ 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 $\mathbf{C} = \mathbf{AB}$. 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 $\mathbf{C} = \mathbf{AB}$. 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.

Algorithm 1: Parents BFS.

```
   Input: A, startVertex
1  Function ParentsBFS
2      p(startVertex) = startVertex
3      q(startVertex) = startVertex
4      for level = 1 to nrows(A) − 1 do
5          q^T⟨¬s(p^T), r⟩ = q^T any.secondi A
6          p⟨s(q)⟩ = q
7          if nvals(q) = 0 then
8              return
```

B. Betweenness Centrality (BC)

The vertex betweenness-centrality metric is a weighted measure of the number of shorted paths that go through any given node,

$$\sum_{s\neq i\neq t} \frac{\sigma(s,t|i)}{\sigma s,t},$$

where $\sigma(s,t)$ is the total number of shortest paths from node s to t, and $\sigma(s,t|i)$ is the total number of shortest paths from node s to t that pass through node i. This is expensive to compute, so in practice, a subset of source nodes s are chosen at random (a *batch*).

Algorithm 2: Direction Optimizing Parent BFS.

```
   Input: A, A^T, startVertex
1  Function DirectionOptimizingBFS
2      q(startVertex) = 0
3      for level = 1 to nrows(A) − 1 do
4          if Push(A, q) then
5              q^T⟨¬s(p^T), r⟩ = q^T any.secondi A
6          else
7              q⟨¬s(p), r⟩ = A^T any.secondi q
8          p⟨s(q)⟩ = q
9          if nvals(q) = 0 then
10             return
```

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

To simplify the presentation of the entire BC algorithm, Alg. 3 does not show the direction-optimization. It is the same transformation as the pair of BFS algorithms, where the push-only step (line 5 of Alg. 1), is expanded to a push/pull hueristic (lines 4-7 of Alg. 2).

C. PageRank (PR)

PageRank (PR) computes the importance of each node as a recursively-defined metric: a web page is important if important pages link to it. Alg. 4 shows the GraphBLAS implemenation of PR as specified in the GAP Benchmark. It uses the plus.second semiring, where $\mathsf{second}(x,y) = y$, so it can ignore any edge weights in the input matrix. The PR in GAP does not properly handle dangling vertices in the graph. The Graphalytics benchmark has a PageRank variant which avoids this problem [15]. We have included this version to compare its performance with the GAP benchmark algorithm `pr.cc`.

Algorithm 3: Betweenness centrality.

Function *BrandesBC*
- // The **NumSp** structure holds the number of shortest paths for each node and starting vertex discovered so far.
- // Initialized to source vertices.
- $\mathbf{NumSp} \leftarrow \{\boldsymbol{s}, [1, 1, \ldots, 1]\}$
- // The **Frontier** holds the number of shortest paths for each node and starting vertex discovered so far.
- $\mathbf{Frontier} \leftarrow \{\boldsymbol{s}, [1, 1, \ldots, 1]\}$
- $\mathbf{Frontier}\langle\neg s(\mathbf{NumSp}), \mathrm{r}\rangle = \mathbf{Frontier}\,\mathsf{plus.first}\,\mathbf{A}$
- // The **Sigmas** matrices store frontier information for each level of the BFS phase.
- // BFS phase (forward sweep)
- **for** $d = 0$ **to** $\mathrm{nrows}(\mathbf{A})$ **do**
  - // $\mathbf{Sigmas}[d](:, s) = d^{\text{th}}$ level frontier from source vertex s
  - let: $\mathbf{Sigmas}[d] \in \mathbb{B}^{n \times nsver}$
  - $\mathbf{Sigmas}[d](:,:) = \mathbf{Frontier}$ // Convert matrix to Boolean
  - $\mathbf{NumSp} \mathrel{+}= \mathbf{Frontier}$ // Accumulate path counts
  - $\mathbf{Frontier}\langle\neg s(\mathbf{NumSp}), \mathrm{r}\rangle = \mathbf{Frontier}\,\mathsf{plus.first}\,\mathbf{A}$ // Update frontier
  - **if** $\mathrm{nvals}(\mathbf{Frontier}) = 0$ **then**
    - break
- let: $\mathbf{BCU} \in \mathbb{Q}_{32}^{n \times nsver}$
- $\mathbf{BCU}(:) = 1.0$ // Make **BCU** dense, initialize all elements to 1.0
- let: $\mathbf{W} \in \mathbb{Q}_{32}^{n \times nsver}$
- // Tally phase (backward sweep)
- **for** $i = d - 1$ **downto** 0 **do**
  - $\mathbf{W}\langle s(\mathbf{Sigmas}[i]), \mathrm{r}\rangle = \mathbf{BCU}\ \mathsf{div}_{\cap}\ \mathbf{NumSp}$
  - $\mathbf{W}\langle s(\mathbf{Sigmas}[i-1]), \mathrm{r}\rangle = \mathbf{W}\ \mathsf{plus.first}\ \mathbf{A}^{\mathsf{T}}$
    - // Add contributions by successors and mask with that BFS level's frontier.
  - $\mathbf{BCU} \mathrel{+}= \mathbf{W} \times_{\cap} \mathbf{NumSp}$
- // Row reduce **BCU** and subtract *nsver* from every entry to account for 1 extra value per **BCU** row element
- $\mathbf{delta}(:) = -nsver$
- $\mathbf{delta} \mathrel{+}= [+_j\, \mathbf{BCU}(:, j)]$

Algorithm 4: PageRank (as specified in the GAPBS).

Data: $\mathbf{A} \in \mathbb{B}^{n \times n}$ // adjacency matrix
damping // damping factor
tol // stopping tolerance
itermax // maximum number of iterations
Result: $\mathbf{r} \in \mathbb{Q}^n$

Function *PageRank*
- $teleport = \frac{1-\alpha}{n}$
- $\mathbf{r}(0 : n-1) = \frac{1}{n}$, $\mathbf{t} = \mathbb{Q}^n$
- $\mathbf{d_{out}} = [+_j\, \mathbf{A}(:, j)]$ // precomputed rowdegree
- $\mathbf{d} = \mathbf{d_{out}}\ \mathsf{div}_{\cap}\ damping$ // prescale with damping
- **for** $k = 1$ **to** $numIterations$ **do**
  - swap $\mathbf{t}$ and $\mathbf{r}$ // t is now the prior rank
  - $\mathbf{w} = \mathbf{t}\ \mathsf{div}_{\cap}\ \mathbf{d}$
  - $\mathbf{r}(0 : n-1) = teleport$
  - $\mathbf{r} \mathrel{+}= \mathbf{A}^{\mathsf{T}}\ \mathsf{plus.second}\ \mathbf{w}$
  - $\mathbf{t} \mathrel{-}= \mathbf{r}$
  - $\mathbf{t} = abs(\mathbf{t})$
  - **if** $[+_j\, \mathbf{t}(:)] < tol$ **then**
    - return

Algorithm 5: SSSP (delta-stepping).

Data:
$\mathbf{A}, \mathbf{A_H}, \mathbf{A_L} \in \mathbb{Q}^{|V| \times |V|}$
$s, i \in \mathbb{N}$
$\Delta \in \mathbb{Q}$
$\mathbf{t}, \mathbf{t_{Req}} \in \mathbb{Q}^{|V|}$
$\mathbf{t_{B_i}}, \mathbf{e} \in \mathbb{N}^{|V|}$

Function *DeltaStepping*
- $\mathbf{A_L} = \mathbf{A}\langle 0 < \mathbf{A} \le \Delta\rangle$
- $\mathbf{A_H} = \mathbf{A}\langle \Delta < \mathbf{A}\rangle$
- $\mathbf{t}(:) = \infty$
- $\mathbf{t}(s) = 0$
- **while** $\mathrm{nvals}(\mathbf{t}\langle i\Delta \le \mathbf{t}\rangle) \neq 0$ **do**
  - $s = 0$
  - $\mathbf{t_{B_i}} = \mathbf{t}\langle i\Delta \le \mathbf{t} < (i+1)\Delta\rangle$
  - **while** $\mathbf{t_{B_i}} \neq 0$ **do**
    - $\mathbf{t_{Req}} = \mathbf{A_L}^{\mathsf{T}}\ \mathsf{min.plus}\ (\mathbf{t} \otimes \mathbf{t_{B_i}})$
    - $\mathbf{e} = \mathbf{t}\langle 0 < \mathbf{e} \oplus \mathbf{t_{B_i}}\rangle$
    - $\mathbf{t_{B_i}} = \mathbf{t}\langle i\Delta \le \mathbf{t_{Req}} < (i+1)\Delta\rangle \otimes (\mathbf{t_{Req}}\ \mathsf{min}_{\cup}\ \mathbf{t})$
    - $\mathbf{t} = \mathbf{t}\ \mathsf{min}_{\cup}\ \mathbf{t_{Req}}$
  - $\mathbf{t_{Req}} = \mathbf{A_H}^{\mathsf{T}}\ \mathsf{min.plus}\ (\mathbf{t} \otimes \mathbf{e})$
  - $\mathbf{t} = \mathbf{t}\ \mathsf{min}_{\cup}\ \mathbf{t_{Req}}$
  - $i = i + 1$

D. Single-Source Shortest-Paths (SSSP)

A Delta-Stepping Single-Source-Shortest-Path algorithm in GraphBLAS is shown in Alg. 5. It relies on the min.plus semiring. Since it is a fairly complex algorithm, refer to [24] for a description of the method.

E. Triangle Counting (TC)

The triangle counting (TC) problem is to compute the number of unique cliques of size 3 in a graph. The TC algorithm is shown in Alg. 6, based on [26]. It starts with a heuristic that decides when to sort the input graph by ascending degree. Next, it constructs the lower and upper triangular part and computes a masked matrix multiply using the plus.pair semiring. Internally, a dot product method is used in SS:GrB, because $\mathbf{U}$ is transposed via the descriptor. The pair is the simple function $\mathsf{pair}(x, y) = 1$. When used in a semiring, it acts like the times operator of the conventional semiring, except that it can ignore the values of its inputs and treat them both

Algorithm 6: Triangle counting

Data: $\mathbf{A} \in \mathbb{B}^{n \times n}$
Result: $t \in \mathbb{N}$
Function *TriangleCount*
  sample the *mean* and *median* degree of $\mathbf{A}$
  if $mean > 4 \times median$ **then**
    $\mathbf{p}$ = permutation to sort degree, ascending order
    $\mathbf{A} = \mathbf{A}(\mathbf{p}, \mathbf{p})$
  $\mathbf{L} = \text{tril}(\mathbf{A})$
  $\mathbf{U} = \text{triu}(\mathbf{A})$
  $\mathbf{C}\langle s(\mathbf{L})\rangle = \mathbf{L}$ plus.pair $\mathbf{U}^\mathsf{T}$
  $t = [+_{ij}\, \mathbf{C}(i,j)]$

as 1. This semiring is useful for structural computations, such as triangle counting, when the edge weights of a graph may be present but should be ignored in a particular algorithm.

F. Connected Components

The connected components algorithm in LAGraph (Alg. 7) is written by Zhang, Azad, and Buluç [30], [31]. The method maintains a forest of trees represented by a parent vector, and interatively merges trees until no more merging is possible. The method as shown in Alg. 7 is a simplified variant that it operates on the entire graph. In the LAGraph version, a subgraph is constructed first, and the method finds the connected components of the subgraph, and then operates on the entire graph.

Algorithm 7: Connected components (FastSV).

Function *FastSV*
  $n = \text{nrows}(\mathbf{A})$
  $\mathbf{gf} = \mathbf{f}$
  $\mathbf{dup} = \mathbf{gf}$
  $\mathbf{mngf} = \mathbf{gf}$
  $\{\boldsymbol{i}, \boldsymbol{x}\} \leftarrow \mathbf{f}$
  repeat
    // Step 1: Stochastic hooking
    $\mathbf{mngf} = \mathbf{mngf}$ min $\mathbf{A}$
    $\mathbf{mngf} = \mathbf{mngf}$ second.min $\mathbf{gf}$
    $\mathbf{f}(\boldsymbol{x}) = \mathbf{f}$ min $\mathbf{mngf}$
    // Step 2: Aggressive hooking
    $\mathbf{f} = \mathbf{f}$ min $\mathbf{mngf}$
    // Step 3: Shortcutting
    $\mathbf{f} = \mathbf{f}$ min $\mathbf{gf}$
    // Step 4: Calculate grandparents
    $\{\boldsymbol{i}, \boldsymbol{x}\} \leftarrow \mathbf{f}$
    $\mathbf{gf} = \mathbf{f}(\boldsymbol{x})$
    // Step 5: Check termination
    $\mathbf{diff} = \mathbf{dup} \neq \mathbf{gf}$
    $sum = [\oplus_i \mathbf{diff}(i)]$
    $\mathbf{dup} = \mathbf{gf}$
  until $sum == 0$

V. Utility Fuctions

Here is a rough categorization of the utilities (not all included yet)

- **Helpers:** TRY/CATCH, MIN/MAX, tic/toc, TypeName, KindName
- **Memory management:** malloc, calloc, realloc, free stuff. This might be covered in the design decisions section
- **Threading:** Get/SetNumThreads. Shouldn't this be part of init, or an LAGraph_Context? Should this also be discussed in design decisions. GraphBLAS threading is one thing, but this seems to be LAGraph threading (outside of GraphBLAS calls). I could see this as algorithm specific and not a general util. The only way to leverage these inside algorithms is to either set a global property that all algorithms have access to, or creating a context that is passed to algorithms.
- **Graph Properties:** If they operate on the LAGraph_Graph cached properties consider a consistent naming scheme like LAGraph_Property_XXX
  - DeleteProperties (consider Property_ClearAll), Property_AT, Property_AssymetricPattern, Property_RowDegree, Property_ColDegree
  - CheckGraph (consider Property_Check) check what?
  - SortByDegree, SampleDegree (maybe belongs grouped here as it is computing properties, but maybe not cached
- **Graph I/O:** BinRead, MMRead (we are missing BinWrite, MMWrite), DisplayGraph (is this a pretty print of the matrix or all the properties, this should also take FILE*)
- **Graph ops:** Pattern (MakePattern, GetPattern). Not sure how to categorize these
- **Graph query**: IsEqual, IsAll. Are these graph or matrix utils? IsAll is an ambiguous (maybe misleading) name because it seems to be a generic comparator (consider CompareGraphs).
- **Consider for removal** Things that may be implementation detail and could be buried:
  - Sort1/2/3 - currently only Sort2 is used. I don't see a strong need to include these as part of the public API at this time.
  - Random15/60 - I see Random64 being the most widely usable. These can easily be buried as well, and I would still suggest Random64 (Random60 seems too SuiteSparse:GraphBLAS specific.
  - Test_ReadProblem (move to the TestArea)

VI. Evaluation

The performance of LAGraph can only be considered in context of an implementation of the underlying GraphBLAS library. This is discussed in Section VI-A, followed by performance results of the new LAGraph API on the 6 algorithms in the GAP Benchmark [4].

A. SuiteSparse Extensions

In a prior paper ([3]), an early draft of SuiteSparse:GraphBLAS v4.0.0 (Aug 2020) was compared with the

GAP benchmark [4] 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 [10], 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 [4], [27]. 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 [3] (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.

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* [12]. 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 and BC.

Another useful addition to SS:GrB is the new positional binary operators, such as the any.secondi used in the BFS.

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 (Feb 7, 2021) and SuiteSparse:GraphBLAS 4.0.4-draft (Feb 7, 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 II 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 III.

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	24.52	30.69	9.11	8.43	34.06
BFS : GAP	.31	.58	.22	.34	.25
BFS : SS	.52	1.22	.33	.66	3.32
PR : GAP	19.81	25.29	15.16	5.13	1.01
PR : SS	22.17	27.71	17.21	9.30	1.34
CC : GAP	.53	1.66	.23	.22	.05
CC : SS	3.36	4.47	1.47	1.97	.98
SSSP : GAP	4.91	7.23	2.02	.81	.21
SSSP : SS	17.37	25.54	8.54	9.61	46.79
TC : GAP	374.08	21.83	79.58	22.18	.03
TC : SS	917.99	34.01	239.58	34.65	.23

TABLE II: 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 III: 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 [3].

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 (1.3x for Kron, 1.5x for Urand, and 1.2x for Twitter).

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 1.5x to 2x slower than the GAP benchmark. We expect the remaining 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 [17].

2) In GraphBLAS, the BFS must be expressed as two calls. The first computes $\mathbf{q}\langle\neg\mathbf{p}\rangle = \mathbf{q}^{\mathsf{T}}\mathbf{A}$, and the second updates the parent vector, $\mathbf{p}\langle s(\mathbf{q})\rangle = \mathbf{q}$:

```
GrB_vxm (q, p, NULL, semiring, q, A, GrB_DESC_RSC) ;
GrB_assign (p, 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 parent vector `p`. 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 for the Road graph, LAGraph+SS:GrB is quite slow for all but PageRank (PR). The primary reason for this is the high diameter of the Road graph (about 6980). This requires 6980 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 $\mathbf{C}\langle s(\mathbf{L})\rangle = \mathbf{L}\mathbf{U}^{\mathsf{T}}$, followed by the reduction of $\mathbf{C}$ to a single scalar. The matrix $\mathbf{C}$ is then discarded. All that GraphBLAS needs is a fused kernel that does not explicitly instantiate the temporary matrix $\mathbf{C}$. This is permitted by the GraphBLAS C API Specification, but not yet implemented in SS:GrB.

VII. Conclusion

Future work - ideas:

- Create a Python wrapper for LAGraph
- Implement the LDBC Graphalytics benchmark [14]
- Improve data ingestion performance using e.g., SIMD instructions [18]

Acknowledgements

TODO: 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.

T. Davis was supported by NSF CNS-1514406, NVIDIA, Intel, MIT Lincoln Lab, Redis Labs, and IBM.

References

[1] "NetworkX," in *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*, R. Alhajj and J. G. Rokne, Eds. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4939-7131-2_100771

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

[3] 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

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

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

[6] 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

[7] 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

[8] ——, "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

[9] A. Buluç *et al.*, "The GraphBLAS C API specification. Version 1.3.0," 2019, https://people.eecs.berkeley.edu/~aydin/GraphBLAS_API_C_v13.pdf.

[10] 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

[11] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.org

[12] T. A. Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, 2019. [Online]. Available: https://doi.org/10.1145/3322125

[13] 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.

[14] 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

[15] A. Iosup, A. Musaafir, A. Uta, A. Prat-Pérez, G. Szárnyas, H. Chafi, I. G. Tanase, L. Nai, M. J. Anderson, M. Capota, N. Sundaram, P. A. Boncz, S. Depner, S. Heldens, T. Manhardt, T. Hegeman, W. L. Ngai, and Y. Xia, "The LDBC Graphalytics benchmark," *CoRR*, vol. abs/2011.15028, 2020. [Online]. Available: https://arxiv.org/abs/2011.15028

[16] 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

[17] 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.

[18] 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

[19] J. Leskovec and R. Sosic, "SNAP: A general-purpose network analysis and graph-mining library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, pp. 1:1–1:20, 2016. [Online]. Available: https://doi.org/10.1145/2898361

[20] 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.

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

[22] ——, "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
[23] 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
[24] U. Sridhar *et al.*, "Delta-stepping SSSP: From vertices and edges to graphblas implementations," in *GrAPL at IPDPS*. IEEE, 2019, pp. 241–250. [Online]. Available: https://doi.org/10.1109/IPDPSW.2019.00047
[25] 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
[26] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.
[27] 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
[28] ——, "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
[29] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan, "GBTL-CUDA: graph algorithms and primitives for gpus," in *GABB at IPDPS*. IEEE, 2016, pp. 912–920. [Online]. Available: https://doi.org/10.1109/IPDPSW.2016.185
[30] Y. Zhang, A. Azad, and A. Buluç, "Parallel algorithms for finding connected components using linear algebra," *Journal of Parallel and Distributed Computing*, vol. 144, pp. 14–27, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731520302689
[31] Y. Zhang *et al.*, "FastSV: A distributed-memory connected component algorithm with fast convergence," in *PPSC*. SIAM, 2020, pp. 46–57. [Online]. Available: https://doi.org/10.1137/1.9781611976137.5