LAGraph: A Community Effort to Collect Graph Algorithms built on top of the GraphBLAS

Tim Mattson[‡], Tim Davis[⋄], Manoj Kumar[¶], Aydın Buluç[†], Scott McMillan[§], José Moreira[¶], Carl Yang[*,†]

[‡]*Intel Corporation* [†]*Computational Research Division, Lawrence Berkeley National Laboratory*
[⋄]*Texas A and M University* ¶*IBM Corporation* §*Software Engineering Institute, Carnegie Mellon University*
[*]*Electrical and Computer Engineering Department, University of California, Davis*

***Abstract*—In 2013, we released a position paper to launch a community effort to define a common set of building blocks for constructing graph algorithms "in the language of Linear Algebra". This led to the GraphBLAS. We released a specification for the C programming language binding to the GraphBLAS in 2017. Since that release, multiple libraries that conform to the GraphBLAS C specification have been produced.**

In this position paper, we launch the next phase of this ongoing community effort; a project to assemble a set of high level graph algorithms built on top of the GraphBLAS. While many of these algorithms are well-known with high quality implementations available, they have not been assembled in one place and integrated with the GraphBLAS. We call this project the LAGraph graph algorithm project and with this position paper, put out a call for collaborators to join us. While the initial goal is to just assemble these algorithms, we hope in the long run this effort will result production worthy code; with the LAGraph library serving as an open source repository of verified graph algorithms that use the GraphBLAS.

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

I. Introduction

Graphs are an essential abstraction for a wide range of problems. There are many ways to represent algorithms over graphs. One class of methods defines graphs in terms of matrices. For example, we can define a graph in terms of an adjacency matrix where the rows and columns are labeled by the vertices and the non-zero elements of the matrix are the edges between vertices.

Expressing graph algorithms in the "language of linear algebra" [1] is a mature subject with multiple high performance graph libraries based on sparse linear algebra [2], [3], [4], [5], [6].

A community of researchers came together to define common building blocks for graphs expressed in the language of linear algebra. They launched this effort with a position paper [7] in 2013 and formed the GraphBLAS Forum [8] to standardize the low-level building blocks used in these graph algorithms. After almost three years of steady work, the GraphBLAS forum completed the mathematical formalizations of GraphBLAS [9]. With another one and a half years of work, the group completed the C language binding to the GraphBLAS math spec [10].

Currently there are multiple implementations of the GraphBLAS C specification. We have learned a great deal about how to define these operations and how to implement them efficiently. We are now ready to launch the next phase of the project: to produce a library of high level graph algorithms implemented on top of the GraphBLAS. We call this library of algorithms *LAGraph*. Just as we launched the GraphBLAS project with a position paper, we are launching this next phase of the project with a position paper.

The goals of the LAGraph effort is first and foremost to bring together the full range of known graph algorithms that can be constructed with the GraphBLAS. From this collection we will be able to systematically asses the coverage of graph algorithms based on Linear Algebra. This will also serve as raw material in ongoing studies of the fundamental design patterns exploited by linear algebra based graph algorithms.

Initially, LAGraph is not intended to be a production-worthy library of high level graph algorithms. We expect over time, however, that from the LAGraph effort, we will produce such a library. Anticipating that goal, we are constructing the library "as if" it was to be used by data analytics end-users; i.e. by people who need the results from graph algorithms with little concern for how they are implemented. This requirement of building a library for *end-users* as opposed to a library for *graph algorithm researchers* has far reaching implications for the design of this software.

We start the paper with a brief summary of the objects and operations defined in the C specification of the GraphBLAS. Next, we describe the repository where we will build LAGraph. This is important since the purpose of a position paper is to attract a community of researchers to join the effort which means we want people to understand how to work with and perhaps contribute algorithms to the repository. We then discuss the challenges we faced in writing the early version LAGraph and what it suggests about future developments needed in the GraphBLAS themselves. We close with concluding remarks.

II. The GraphBLAS

Consider a graph represented as an n-by-n adjacency matrix $\mathbf{A}$, where A_{ij} is the weight of the edge from vertex i to vertex j, and a second k-by-n matrix $\mathbf{B}$ representing a subset (of size k) of the vertices in the graph, such that B_{ji} is 1 if the jth element of the subset is vertex i (and all other elements of $\mathbf{B}$ are 0). The traditional matrix product $\mathbf{B} \times \mathbf{A}$ over real arithmetic of these two matrices returns the cost based on the edge weights of reaching the set of vertices adjacent to the vertices in $\mathbf{B}$. This fundamental operation can be used to construct a wide range of graph algorithms.

We extend the range of graph operations by keeping the basic pattern of a matrix-matrix multiplication, but varying the operators and the interpretation of the values in the matrices (the *domain*). By carefully choosing operators and the domain, we control the relation between matrix operations familiar in linear algebra and graph operations, thereby enabling composable graph algorithms.

In addition to matrix multiplication, the GraphBLAS math specification defines a range of additional operations over matrices and vectors. These are summarized in Table I.

TABLE I: A mathematical overview of the fundamental GraphBLAS operations supported in this specification. $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are GraphBLAS matrices; $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are GraphBLAS vectors; i and j are single indices; $\mathbf{i}$ and $\mathbf{j}$ are arrays of indices; $\oplus$ and $\otimes$ are arbitrary element-wise operators; the element-wise $\odot$ operator is used for the optional accumulation with the output GraphBLAS object where $x \odot= y$ implies $x = x \odot y$; and $F_u()$ is a unary function. Although not shown here, the input matrices $\mathbf{A}$ and $\mathbf{B}$ may be selected for transposition prior to the operation, and masks can be used to control which values are written to the output GraphBLAS object.

Operation name	Mathematical description
mxm	$\mathbf{C} \odot= \mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{w} \odot= \mathbf{A} \oplus . \otimes \mathbf{v}$
vxm	$\mathbf{w}^T \odot= \mathbf{v}^T \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C} \odot= \mathbf{A} \otimes \mathbf{B}$
	$\mathbf{w} \odot= \mathbf{u} \otimes \mathbf{v}$
eWiseAdd	$\mathbf{C} \odot= \mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w} \odot= \mathbf{u} \oplus \mathbf{v}$
reduce (row)	$\mathbf{w} \odot= \bigoplus_j \mathbf{A}(:,j)$
apply	$\mathbf{C} \odot= F_u(\mathbf{A})$
	$\mathbf{w} \odot= F_u(\mathbf{u})$
transpose	$\mathbf{C} \odot= \mathbf{A}^T$
extract	$\mathbf{C} \odot= \mathbf{A}(\mathbf{i},\mathbf{j})$
	$\mathbf{w} \odot= \mathbf{u}(\mathbf{i})$
assign	$\mathbf{C}(\mathbf{i},\mathbf{j}) \odot= \mathbf{A}$
	$\mathbf{w}(\mathbf{i}) \odot= \mathbf{u}$

In mapping the GraphBLAS as a set of mathematical operators onto the C programming language we made a number of fundamental choices [10]. First, the core data structures required to represent the objects defined by the GraphBLAS are opaque. The GraphBLAS API defines a contract with the programmer for how these objects will be used, but the implementations and underlying data structures are left to the implementation. This opaqueness is critical if the API is to serve diverse hardware ranging from CPUs to GPUs to specialized graph hardware. Second, we defined a non-blocking execution model that allows lazy evaluation. Ultimately, to optimize sparse linear algebra software we need to aggressively fuse operations and even restructure algorithms. This requirement meant that we had to be very careful to define when results from a sequence of GraphBLAS operations must be materialized.

Since the release of the GraphBLAS specification, several implementations of the GraphBLAS have been developed.

A. SuiteSparse:GraphBLAS

SuiteSparse:GraphBLAS is the first full implementation of the GraphBLAS standard, and is available at http://suitesparse.com [11]. The design of a GraphBLAS library is flexible, because its data structures are opaque to the user. SuiteSparse:GraphBLAS uses a compressed-sparse vector data structure, in four different forms. A matrix can be stored in row-major order (CSR), or column-major order (CSC). Each sparse vector consists of a sorted list of indices, and the corresponding numerical values. The sparse vectors are packed together into two arrays, and another "pointer array" (of size equal to the dimension of the matrix, say n) keeps track of where each row (or column) starts. The memory taken is $O(n+e)$ for a CSR matrix with n rows or a CSC matrix with n columns, and with e entries. Most graphs have $e = O(n)$ entries, but some graphs (and in particular, subgraphs) can be *hypersparse* [12], with $e << n$. In the hypersparse form, the pointer array itself becomes sparse, and non-empty vectors take no space at all. The space is reduced to $O(e)$, so that matrices with enormous dimensions can be created, as long as $e << n$. SuiteSparse:GraphBLAS exploits hypersparsity automatically, and all methods can operate on all four matrix formats in any combination.

The ability to incrementally modify a graph is critical in many applications. GraphBLAS includes two operations that can make small incremental changes to a graph/matrix: namely `GrB_setElement` and `GrB_assign`. It would be exceedingly slow to insert or delete a single entry in a CSR or CSC format, taking $O(n+e)$ time **per entry** inserted or deleted. Instead, the non-blocking aspect of GraphBLAS is exploited. Fast deletion of entries is handled by creating *zombies*, which are entries tagged for later deletion. Fast insertion is handled with *pending tuples*, which is a separate unordered list of (i, j, a_{ij}) for each new entry. When a matrix operation occurs (such as matrix multiply), all zombies are killed and all pending tuples are assembled, in a single $O(n+e+p \log p)$ step (for p pending tuples), or $O(e + p \log p)$ in the hypersparse case. As a result, it is just as fast to use a sequence of e `GrB_setElement` operations to build a matrix, as it is to create an array of e tuples and use `GrB_build`. Internally, SuiteSparse:GraphBLAS is building the list itself, for the user, and then does a `GrB_build` when the matrix needs to be completed.

To enable high-performance matrix-matrix multiply, a code generation mechanism is used to build functions for each semiring that can be created with built-in operators. The functions can rely on Gustavson's method [13], a dot product method, and heap-based method, all with masked variants. A current prototype of the package adds an early exit mechanism for the MIN, MAX, OR, and AND monoids, where a dot product can terminate as soon as a terminal value is found in the result (`true` for OR, for example). This will enable a fast direct-optimizing BFS to be written, where the "pull" is a dot product, and the "push" a saxpy-based operation (Gustavson's or the heap method).

Since its implementation was requested as the GraphBLAS reference implementation, testing is a vital component to the package. In SuiteSparse:GraphBLAS, each GraphBLAS operation was written twice: once in high-performance algorithms in C, and again in a very simple and short MATLAB script, using dense matrices with of the required type. For example, `GrB_assign` requires about 3,908 lines of C (not counting comments), but only 161 lines in MATLAB. The pattern in the MATLAB version is held as a separate Boolean matrix. The MATLAB functions are not intended to be fast. Instead, they exactly mimic the GraphBLAS API Specification, line by line, so they can be visually inspected for conformance to the spec. For example, matrix multiply is written with a triply-nested `for` loop. Then, to test the package, each computation is done in SuiteSparse:GraphBLAS (via a MATLAB interface) and in the MATLAB mimic. The tests pass only if the results are identical in both value and pattern (even with identical floating-point roundoff error, in most cases).

SuiteSparse:GraphBLAS appears in Debian and Ubuntu Linux distros, and has been released as part of the RedisGraph database module of the Redis database systems, by RedisLabs, Inc. (https://oss.redislabs.com/redisgraph/).

B. IBM GraphBLAS

TODO: Manoj and Jose: Describe the IBM GraphBLAS implementation

C. Gunrock GraphBLAS

TODO: Carl: Describe the GunRock implementation

D. GBTL: GraphBLAS Template Library

TODO: Scott, Tim M and Aydin: Describe C++ libraries consistent with the GraphBLAS such as GBTL and CombBLAS. Describe Python binding.

III. LAGraph Repository

The hypothesis underlying the GraphBLAS is that algorithm designers can focus on expressing their algorithms in terms of the high level linear algebra operations defined in the GraphBLAS while leaving low level optimizations to any particular hardware platform to the implementor of the GraphBLAS. Ultimately, we want hardware vendors to be responsible for creating highly tuned versions of the GraphBLAS specialized to the features of their systems.

Algorithm designers will naturally wonder how much performance is lost due to the use of a high level API such as the GraphBLAS. We believe the GraphBLAS bring inherent efficiency advantages to a graph algorithms due to the more structures access to data afforded by the sparse linear algebra formats. This is an untested hypothesis, however, since until now, we have not had multiple implementations of a high level API tuned to the features of a range of platforms.

Testing this hypothesis of the performance potential afforded by the GraphBLAS is a major outcome we anticipate from the LAGraph project. By collecting high level graph algorithms and validating them across an engaged community, we will produce the library of algorithms needed to evaluate the effectiveness of the GraphBLAS approach.

Before we can conduct such experiments, however, we need to collect graph algorithms implemented on top of the GraphBLAS. We have created a git hub repository at https://github.com/GraphBLAS/LAGraph for members of the LAGraph community to use to contribute GraphBLAS algorithms. The basic elements of the repository include:

- A Build System for creating the LAGraph library and the test routines.
- A library of utilities including loading matrices from disk in Matrix Market format [14], evaluating results, and creating random test matrices.
- A directory of graph algorithms.
- A directory holding a test harness for each algorithm.

We will write documentation, a programmer's reference guide and define procedures for how people can add new algorithms.

IV. Discussion

We are early in the LAGraph project. At this point, we've defined the basic structure of the repository and the overall goals of the project. We have built an early framework for testing and core utility routines to support software development. Finally, we assembled a few algorithms which we are using to test the basic structure of the software system.

Even at this early phase of the project, we have learned a great deal about how the GraphBLAS will interact with end-users. The objects manipulated by the GraphBLAS are opaque. An implementation needs complete freedom in how data structures underlying the GraphBLAS are implemented. A graph algorithm, however, is used as part of a processing pipeline. For example, data may exist in data frames. A subset of the data is collected and filtered to produce relationships represented by a graph. Properties of the graph are computed and based on the result a new branch in the processing pipeline may be accessed.

The key here is that graphs inside the GraphBLAS are opaque, but externally they are anything but opaque. This suggests that we need to define functions to import and export data in standard sparse array formats into LAGraph. The initial thinking was that this import functionality would be part of LAGraph and not the GraphBLAS. The only way to do that, however, is if we repacked the input sparse format into separate arrays for column indices, row indices, and values and then use `GrB_build_matrix` to construct the GraphBLAS matrix object. This would extremely inefficient. We need a way to directly import arrays in standard sparse formats into the GraphBLAS and since the GraphBLAS data types are opaque, this can only be done as a GraphBLAS routine.

Graph algorithms do not occur in isolation. The LAGraph library, therefore, needs to return a handle to an opaque GraphBLAS object so it can be used without incurring copy overhead in subsequent graph operations. Given the nonblocking execution model, this raises interesting design questions about how memory consistency between the library and the application is managed.

Graphs can be quite large. Hence, we do not believe the default mode should be that sparse arrays input to LAGraph are copied into a separate memory region to hold the opaque GraphBLAS object. We believe it is important that the memory for the input array be used to hold the GraphBLAS object. This means the input array is "destroyed" (from the perspective of the external library or user application) and replaced with the GraphBLAS opaque object. Since during a computation it is common that elements of a matrix fill-in, the GraphBLAS routines will in some cases need to allocate additional memory during a calculation. This may require deallocating memory and reallocating a larger block of memory. This means we will likely encounter situations where a region of memory allocated in an application is deallocated inside a library routine. This violates the separation of concerns between application and library code expected in well engineered software. There is also the question of communicating to the library routine how the input sparse array was allocated in the first place so the right deallocator can be used.

A draft of SuiteSparse:GraphBLAS includes a working and fully-tested implementation of the import/export feature, using a strategy much like the "move constructor" of C++. For the export of a CSC matrix, for example, three arrays are removed from the GraphBLAS matrix `A`: a pointer array `Ap` of size `n+1`, an index array `Ai` of size `e`, and a values array `Ax`. The row indices of the `j`th column of the matrix appear as the list `Ai[Ap[j]...Ap[j+1]]`, and the values are in the same locations in `Ax` [15].

The remains of the GraphBLAS object `A` are then deleted, but all of its content is now "owned" by the external library (LAGraph, say), which is then responsible for freeing these three arrays. Assuming that the opaque GraphBLAS object `A` is already in the CSC format, the export takes just $O(1)$ time, and no new memory is allocated. The external library now has access to the graph. If the GraphBLAS library does not support the CSC format internally, it can allocate these arrays, populate them, and then free `A`. The effect is the same; only the performance differs.

The import is symmetric with the export: LAGraph (or any other external library) passes in the three arrays `Ap`, `Ai`, and `Ax`, which are then either incorporated as-is into the `GrB_Matrix A` (taking $O(1)$ time), or copied and freed. Either way, the three arrays are now owned by GraphBLAS, not the external library, and would be freed at some point no later than `GrB_free(&A)`. Since the matrix `A` is opaque, the GraphBLAS library can select whatever method it chooses to take ownership of `Ap`, `Ai`, and `Ax`. It may choose later to `realloc` them if the number of entries needs to grow.

After an export of `A`, and then an import of the same arrays, the GraphBLAS matrix `A` is perfectly reconstructed, ideally in a total of $O(1)$ time. SuiteSparse:GraphBLAS supports the import/export of all four of its formats: CSR, CSC, and their hypersparse variants.

A `malloc` of these `Ap`, `Ai`, and `Ax` arrays by an external library followed by a freeing of the same space inside GraphBLAS with `GrB_free(&A)` requires both libraries to agree on using the same `malloc/free` routines. To do this, the GraphBLAS API would need to be augmented to allow an external library to select which `malloc/free` routines should be used.

The point of these issues is that in designing an effective library, there are a host of complicated issues to resolve. A major part of the research contribution of this project will be how we solve this complicated problems.

V. Conclusion

The GraphBLAS forum started its work on standard building blocks for graphs in the language of linear algebra in 2013 [7]. We now have a C specification for the GraphBLAS [10] and multiple implementations [11], [16]. The next step in this journey is to define a library of high level graph algorithms that are based on the GraphBLAS.

The GraphBLAS was a community effort launched by a position paper. This is a position paper to launch LAGraph, our community effort to collect and validate a set of high quality graph algorithms that run on top of the GraphBLAS. We urge readers interested in joining us as we work on LAGraph to contact any of the authors of this position paper.

Acknowledgments

Tim Davis would like to acknowledge the support of SuiteSparse:GraphBLAS by MIT Lincoln Laboratory, Intel, and the National Science Foundation (NSF CNS-1514406).

TODO: Aydın Buluç Carl Yang and maybe Scott will need to acknowledge someone.

References

[1] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011, vol. 22.

[2] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *The Intl. Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496 – 509, 2011.

[3] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for NoSQL databases," in *Intl. Parallel & Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 2015, pp. 822–830.

[4] K. Ekanadham, W. P. Horn, M. Kumar, J. Jann, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and H. Yu, "Graph Programming Interface (GPI): A linear algebra programming model for large scale graph computations," in *Proc. ACM Intl. Conference on Computing Frontiers*, ser. CF '16. New York, NY, USA: ACM, 2016, pp. 72–81.

[5] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[6] S. Che, B. M. Beckmann, and S. K. Reinhardt, "Programming GPGPU graph applications with linear algebra building blocks," *Intl. Journal of Parallel Programming*, pp. 1–23, 2016.

[7] T. Mattson, D. Bader, J. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo, "Standards for graph algorithm primitives," in *High Performance Extreme Computing Conf. (HPEC)*. IEEE, 2013.

[8] "The GraphBLAS Forum," http://graphblas.org/.

[9] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. Owens, C. Yang, M. Zalewski, and T. Mattson, "Mathematical foundations of the GraphBLAS," in *IEEE High Performance Extreme Computing (HPEC)*, 2016.

[10] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *Graph Algorithms Building Blocks workshop at IPDPS (GABB)*. IEEE, 2017.

[11] T. A. Davis, "Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Software*, 2019 (under submission), see http://suitesparse.com.

[12] A. Buluç and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–11.

[13] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, 1978.

[14] R. F. Boisvert, R. Pozo, and K. A. Remington, "The Matrix Market exchange formats: initial design," Tech. Rep. NISTIR 5935, 1996.

[15] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Philadelphia, PA: SIAM, 2006.

[16] "IBM's GraphBLAS implementation," http://graphblas.org/.