GraphBLAS C API: Ideas for future versions of the specification

Timothy G. Mattson*, Carl Yang†, Scott McMillan‡, Aydın Buluç§, José E. Moreira¶

*Intel Corporation, Hillsboro OR, USA (timothy.g.mattson@intel.com)
†University of California, Davis CA, USA (ctcyang@ucdavis.edu)
‡Software Engineering Institute, CMU, Pittsburgh PA, USA (smcmillan@sei.cmu.edu)
§Lawrence Berkeley National Laboratory, Berkeley CA, USA (abuluc@lbl.gov)
¶IBM Corporation, Yorktown Heights NY, USA (jmoreira@us.ibm.com)

***Abstract*—The GraphBLAS C specification provisional release 1.0 is complete. To manage the scope of the project, we had to defer important functionality to a future version of the specification. For example, we are well aware that many algorithms benefit from an inspector-executor execution strategy. We also know that users would benefit from a number of standard predefined semirings as well as more general user-defined types. These and other features are described in this paper in the context of a future release of the GraphBLAS C API.**

I. Introduction

The GraphBLAS aims to standardize the mathematical concepts [4] and the application programming interface (API) [2] for performing graph computations in the language of linear algebra [5]. The GraphBLAS C API version 1.0 has been provisionally released in May 2017 [3]. Its finalization is pending on the existence of at least two feature compliant implementations.

During the C API development process, the members of the GraphBLAS community at large have provided valuable feedback and additional features they would like to see. Some of these ideas have made it to the version 1.0 release, some have been deemed out of scope, and some have been postponed for future releases despite being relevant for GraphBLAS. We also opted to not include many interesting ideas of our own into the first official release due to time constraints and the lack of broad discussion on their implications. This paper presents a high-level overview of those ideas that we are considering for future versions of the GraphBLAS C API.

The topics we describe in this paper are likely to become part of a future release of the C API Spec. We describe standard definitions that aim to ease the burden on the programmer in Section III. An inspector-executor interface that allow preprocessing to be performed on the inputs (matrices, vectors, and masks) of the operation with the aim of accelerating the overall operation is discussed in Section IV. We describe finer grained synchronization constructs in Section V, extensions to descriptor objects in Section VI, user-defined types in Section VII, and additional mathematical operations Section VIII. Finally, Section IX includes a discussion about object type promotion that allows automatic generation of higher dimensional objects from lower dimensional ones.

II. The GraphBLAS C API

This section gives a brief overview of important elements of Version 1.0 of the GraphBLAS C API specification [2] in order give some context for future extensions proposed in this paper. The full specification can be downloaded from the GraphBLAS Forum website [1]. Like the original BLAS, this library performs operations on matrices and vectors. However, there are significant differences to specifically support graph operations as explained in the mathematical specification [4].

Since nearly all operations in the API follow the same basic structure, we will highlight important aspects of the API by examining one operation: matrix-matrix multiply as shown in Figure 1. Figure 1(a) illustrates the mathematical notation for that operation (optional items in brackets). This is similar to what is described in the mathematical specification [4], with some additions supported by the API – write masks and accumulation – which are contained in the signature in Figure 1(b). In the signature for every GraphBLAS operation, the output argument always appears first. This is followed by the write mask and accumulate operation, if supported. Next, the arguments describing the input objects and the math to be performed are listed. Finally, the last argument is the optional descriptor. All GraphBLAS methods return a status

$$\mathbf{C}\langle\mathbf{M}\rangle \;=\; [\mathbf{C} \odot]\; \mathbf{A}^{[T]} \oplus . \otimes \mathbf{B}^{[T]}$$

(a) The mathematical definition.

```
GrB_Info GrB_mxm(GrB_Matrix               C,
                 const GrB_Matrix         M,
                 const GrB_BinaryOp       accum,
                 const GrB_Semiring       op,
                 const GrB_Matrix         A,
                 const GrB_Matrix         B,
                 const GrB_Descriptor     desc);
```

(b) The mxm function signature.

Fig. 1. GraphBLAS matrix-matrix multiply definition and function signature where $\odot$ (corresponding to accum) performs an optional accumulate operation with the output matrix, and $\mathbf{M}$ is an optional write mask that controls which elements of the output matrix get modified.

code (GrB_Info) regarding the success of the call.

A. Objects

At its foundation, the GraphBLAS C API is built on opaque data types exposed through the API by handles. A few of these opaque types are collections – matrices and vectors – where matrices are typically very sparse and vectors can be sparse as well. The mxm signature consists of one output matrix, C, and two input matrices, A and B. The dimensions and domain (type) of the object are specified at construction time and remain fixed for the lifetime of the object. A number of domains corresponding to the C built-in types are supported, and the API allows for definition of user-defined types as well.

The GraphBLAS C API also has opaque types for algebraic objects including unary and binary operators, monoids, and semirings, which differ from the traditional linear algebra of the BLAS. The traditional arithmetic addition and multiplication operations can be replaced (e.g., min and plus, respectively) through user-defined capabilities that can operate on data from multiple domains. The op argument in Figure 1 corresponds to a GraphBLAS semiring, denoted by $\oplus.\otimes$ in the mathematical description, and it specifies the binary operations that take the place of arithmetic addition and multiplication, respectively. The accum argument in the signature, denoted by $\odot$, is an optional binary operator that can be used to combine the results with existing values in the output matrix. If accumulation is not desired, GrB_NULL is specified.

The API supports a few optional control objects: write masks and descriptors. A mask is specified using GraphBLAS matrix (or a vector for 1-dimensional operations). Since this is a write mask, it is applied when assigning results to the output matrix *after* accumulation has been performed. The descriptor is a lightweight object that can modify the semantics of the operation by allowing the user to specify which input parameters need to be transposed prior to the operation, whether the mask should be structurally complemented, or whether the output should be cleared before assignment. Note that GrB_NULL can be specified if default behavior is desired.

B. Operations

The API supports the basic operations that were outlined in mathematical specification [4]: inserting and extracting data, matrix multiply, element wise operations, subgraph assignment and extraction, apply, reduction, and transpose. It also provides a number of variants that are useful in a number of graph algorithms. For example, instead of just row reduction of a matrix to a vector, the API also supports reduction of both matrices and vectors to scalars. Variants of both subgraph assignment and extraction support these operations on a single row or column of a matrix. The API also supports the ability to assign a constant scalar value to an entire subgraph (through a variant of subgraph assignment).

C. Execution Models

The API also supports two execution modes: blocking and nonblocking. In blocking mode, each API method completes the operation before proceeding to the next. However, because the API is meant to support high-performance computing in a parallel and distributed environment we also provide a non-blocking mode. In this mode, the methods *may* return immediately after the input arguments have been verified but before any computation has been carried out. This mode gives an implementation the flexibility to choose an execution strategy that might reduce computation time through fused operations, lazy evaluation, asynchronous evaluation, and/or asynchronous execution. When using nonblocking mode, a slightly different mechanism is used for reporting errors which is beyond the scope of this paper.

III. Standard Definitions

The first way in which the current GraphBLAS specification could be augmented is with a list of pre-defined objects, such as descriptors, semirings and monoids, that are commonly used in building graph algorithms. These standard pre-defined objects would simplify coding and ensure more consistency across algorithms. We emphasize that individual application writers are always free to create their own set of pre-defined objects.

Additionally, GraphBLAS could include pre-defined macros that support shorter calling syntax for methods, by hiding default arguments. Again, individual application writers are always free to create their own set of pre-defined macros.

We propose that such pre-defined objects and macros would be included in a compilation unit by including a header file as follows:

```
#include <GrB_stddef.h>
```

Table I shows an initial set of proposed standard object definitions. It includes descriptors for transposing either or both input matrices, Boolean monoids, the standard arithmetic monoids for integer and floating-point data, and min and max monoids for floating-point data. It also includes a number of commonly used semirings, including the conventional Boolean semiring, arithmetic semirings for integer and floating-point data, as well as min-plus and max-times semirings. Other semirings commonly used with graph algorithms can also be added, and experience with the current API will help in determining which of these are most common.

IV. Inspector-Executor

In its non-blocking mode of execution, GraphBLAS allows specific implementations to support various alternative execution approaches. In particular, it allows for methods to be executed in several stages, possibly interleaved with stages from other methods. This is called *split execution*. In a particular split execution, a method can go through an *analyze* stage followed by a *perform* stage. The analyze stage computes various characteristics of the object being produced, whereas the perform stage does the actual calculations.

It may be desirable to augment GraphBLAS with additional constructs to control this particular analyze/perform split explicitly. In particular, having the application communicate

TABLE I
INITIAL SET OF PROPOSED STANDARD DEFINITIONS FOR GRAPHBLAS.

Name	Description
Descriptors:	
GrB_TA	GrB_INP0 = GrB_TRAN
GrB_TB	GrB_INP1 = GrB_TRAN
GrB_TATB	GrB_INP0 = GrB_INP1 = GrB_TRAN
GrB_TASR	GrB_INP0 = GrB_TRAN, GrB_MASK = GrB_SCMP, GrB_OUTP = GrB_REPLACE
GrB_SR	GrB_MASK = GrB_SCMP, GrB_OUTP = GrB_REPLACE
GrB_R	GrB_OUTP = GrB_REPLACE
Monoids:	
GrB_Lor	$\langle$bool, $\vee$, false$\rangle$
GrB_Land	$\langle$bool, $\wedge$, true$\rangle$
GrB_FP32Min	$\langle$float, min, $\infty\rangle$
GrB_FP32Max	$\langle$float, max, $-\infty\rangle$
GrB_Int32Add	$\langle$int32_t, $+, 0\rangle$
GrB_Int32Mul	$\langle$int32_t, $\times, 1\rangle$
GrB_FP32Add	$\langle$float, $+, 0.0\rangle$
GrB_FP32Mul	$\langle$float, $\times, 1.0\rangle$
GrB_FP64Add	$\langle$double, $+, 0.0\rangle$
GrB_FP64Mul	$\langle$double, $\times, 1.0\rangle$
Semirings:	
GrB_Boolean	$\langle$bool, bool, bool, $\vee, \wedge$, false$\rangle$
GrB_Int32AddMul	$\langle$int32_t, int32_t, int32_t, $+, \times, 0\rangle$
GrB_FP32AddMul	$\langle$float, float, float, $+, \times, 0.0\rangle$
GrB_FP64AddMul	$\langle$double, double, double, $+, \times, 0.0\rangle$
GrB_FP32MinPlus	$\langle$float, float, float, min, $+, \infty\rangle$
GrB_FP64MinPlus	$\langle$double, double, double, min, $+, \infty\rangle$
GrB_FP32MaxTimes	$\langle$float, float, float, max, $\times, -\infty\rangle$
GrB_FP64MaxTimes	$\langle$double, double, double, max, $\times, -\infty\rangle$

properties of an object that do not change between multiple perform stages greatly simplifies or eliminates the need for analyze stages.

As a concrete example, the current GraphBLAS specification exposes mxm as a single operation to the user. Under the hood, it is understood that mxm is often implemented in two phases: analyze and compute. The analyze phase consists of allocating memory to the output matrix. The compute phase computes the value at each nonzero of the output matrix. In this scenario, the implementer has the option of deciding whether to set the nonzero structure of the output matrix in the analyze phase or the compute phase.

It may be desirable in some circumstances to expose these two phases of mxm to the user explicitly. There are a few options on how to do that, including separate methods and leveraging the functionality of the descriptor (see Section VI.)

For typical use cases, neither the size nor the nonzero structure of the output matrix are known *a priori* before the mxm operation is run. However, there may exist situations where the user is computing a sequence of output matrices which share the same nonzero structure and memory layout. In this case, it may be advantageous for the user to explicitly call both the analyze and compute phases for the first matrix-multiplication in the sequence, and only the compute for the rest.

TABLE II
EXPERIMENTAL SETUP USED FOR BENCHMARKING.

Vendor	Intel	Nvidia
Family	Xeon	Tesla GPU
Device	E5-2637 v2	K40c
Codename	Sandy Bridge	Kepler GK110
Memory	256 GB	12 GB
OS	Ubuntu 14.04.1	Ubuntu 14.04.1
Compiler	Intel C++ v17.0.4 Intel OpenMP v5.0	nvcc v8.0.44 g++ v4.9.4
Library API	Intel MKL 2017.4.196 mkl_Scsrmultcsr	cuSPARSE v8.0.44 cusparseScsrgemm2

TABLE III
DESCRIPTION OF DATASETS USED FOR BENCHMARKING. FLOPS REFERS TO THE NUMBER OF FLOATING-POINT COMPUTATIONS REQUIRED TO SQUARE THE MATRIX BY ITSELF. TWO COPIES OF THE MATRIX ARE KEPT IN MEMORY FOR SAKE OF BENCHMARKING.

Dataset	Rows	Nonzeros	Flops	Symmetric
epidemiology	526k	2.1M	16.8M	no
wind tunnel	218k	11.6M	1.3B	yes
protein	36k	4.3M	1.1B	yes
sphere	83k	6.0M	0.9B	yes
hood	221k	10.8M	1.1B	yes
webbase	1.0M	3.1M	139M	no

To further motivate this proposed capability, we used two experimental testbeds to measure the importance of *split execution* for mxm operation. Experiments were performed on a quad-core Intel Xeon CPU E5-2637 v2 @ 3.50GHz with 256 GB RAM running Ubuntu 14.04.1, and an NVIDIA K40c GPU with 12 GB RAM. For each platform we used library (MKL and cuSPARSE, respectively) that supports this type of split execution. Complete details of the test setups are listed in Table II.

For our datasets (see Table III), we chose six matrices from the University of Florida Sparse Matrix Collection [**?**] because they come from diverse sources (epidemic modeling, finite element modeling, protein data, web connectivity) and have varying sparsity structures ranging from fairly regular (*epidemiology*, *wind tunnel*, *protein*, *sphere*) to highly irregular (*hood*, *webbase*). To test the above mentioned use case, we compared calling mxm ten times with no split execution, to a split execution where a call to mxm analyze phase is followed by ten calls to mxm compute phase.

Performance of the test cases is shown in Figure 2. We observe that the split execution performs better than the "unsplit" execution. MKL sees a geometric mean 37.2% speedup going to split execution, while cuSPARSE sees a geometric mean 29.3% speedup. In general, cuSPARSE performed better than MKL on the larger datasets (*wind tunnel*, *protein* and *sphere*), but worse on the smaller ones (*epidemiology* and *webbase*).

We decided to defer such facilities from version 1.0 of the GraphBLAS specification. There is a computational benefit to the user in being able to access split execution, but the API will need to become more complex in order to support it. We

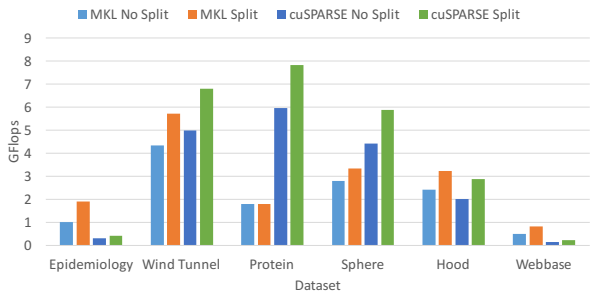

Fig. 2. Comparison of mxm runtime with and without split execution for two experimental setups, MKL and cuSPARSE.

believe that additional experience with implementation and use of GraphBLAS is necessary before we can define the proper interfaces for explicit split execution.

V. Dependency DAGs and wait on Specific Objects

In the current specification of GraphBLAS, when operating in *nonblocking mode*, the operation wait will ensure that the current sequence of GraphBLAS operations has either completed successfully, or encountered an error. Further, a sequence in nonblocking mode where every GraphBLAS operation is followed by an wait call is equivalent to the same sequence in blocking mode with wait calls removed.

It may be desirable to modify the wait interface such that it takes a GraphBLAS object as parameter, and performs all outstanding computations needed to compute that particular object. This approach gives the programmer finer grained specification of the synchronization, while giving the GraphBLAS implementation more flexibility in scheduling operations.

We decided to defer such an interface from version 1.0 of the GraphBLAS specification. We believe that additional experience with implementing and using GraphBLAS nonblocking mode is necessary before we can determine what is the best approach.

VI. API-transparent Extensions

A defining characteristic of the GraphBLAS C API is its use of *descriptors* to modify the behavior of a method. In the current definition of the API descriptors can specify preprocessing steps for input data (typically, whether an input matrix should be transposed or not before the operation) as well as controlling how the result value is written to the output object (complement or not the mask, preserve or replace the elements of the output object that are masked).

This descriptor-based approach allows one to extend the functionality of methods without changing their interface, since all main GraphBLAS methods already include an optional descriptor as the last argument. We envision some new uses for descriptors in future versions of the GraphBLAS C API.

First, we plan to provide a GrB_STRUCTURE_ONLY modifier for masks. In the present specification of GraphBLAS, masks need to be matrices or vectors of a predefined type. And only those elements of a mask that evaluate to a Boolean true value are considered active. Specifying GrB_STRUCTURE_ONLY in the GrB_MASK field of a descriptor would direct GraphBLAS to consider all elements present in the mask as active, irrespective of their value. As a side effect, vectors and matrices of any type could be used as masks.

Another item under consideration is the use of assertions about the properties of the output object. Those assertions could be used to implement optimizations for certain operations. For example, specifying GrB_FIXED_STRUCTURE in the GrB_OUTP field of a descriptor would assert that the output object will not change structure (pattern of nonzero elements) during this operation. Therefore, computation can happen in place with new values simply replacing old values. This would accomplish similar optimization results as achieved by a split analyze/compute execution. Moreover, a spectrum of standard properties, from more restrictive to more permissive can be specified. For example, GrB_SYMMETRIC can assert that an output is a symmetric matrix, whereas GrB_FIXED_NVALS can assert that the number of elements in the output is constant (even though the structure may change).

In the case of assertions, it is important to specify how *hard* or *soft* those assertions are. That is, do they cause a run-time error if violated or are they just "hints" that the implementation can use to improve performance but should be able to recover from, maybe with big performance penalties, if they prove to be wrong.

VII. Generalized User-Defined Types

Currently, GraphBLAS only supports a limited form of user-defined types – the elements that are stored in matrices and vectors. In particular, objects of the data type must have a flat memory representation, so that an object can be copied with a simple `memcpy` operation. It is desirable to lift this restriction. One possibility would be to add a version of GrB_Type_new that supports arbitrary user-defined types as follows.

When the user creates a new type, he or she must pass three functions that perform the most basic operations in that type:

1) A *create* function that creates an object of the user-defined type. That includes allocating storage for the object and initializing that object to a default state.
2) A *copy* function that copies the state of a source object of the user-defined type to a target object of the same user-defined type.
3) A *destroy* function that destroys an object of the user-defined type, releasing any resources the object uses.

Optionally, we could allow create, copy and destroy methods for arrays of user-defined objects, in order to avoid the overhead of function calls and memory management at the level of each individual object. The description of this new form of GrB_Type_new is shown in Figure 3.

VIII. Kronecker Product

One of the proposed operations in the mathematical specification of GraphBLAS is Kronecker product. It is not present in

a) Syntax:

```
GrB_Info GrB_Type_new(GrB_Type    *utype,
                      void        *create,
                      void        *destroy,
                      void        *copy);
```

b) Parameters:

utype (INOUT) On successful return, contains a handle to the newly created user-defined GraphBLAS type object.

create (IN) A pointer to a function that creates and initializes (to a default state) an object of the user-defined type. Such function must return a `void*` pointer to the new object. Its signature is `void* create()`.

destroy (IN) A pointer to a function that destroys an object of the user-defined type, releasing any resources the object uses. Its signature is `void destroy(void* obj)`.

copy (IN) A pointer to a function that copies the contents from a source object of the user-defined type to a destination object of the same user-defined type. Its signature is `void copy(void* tgt, const void* src)`.

c) Return Values:

GrB_SUCCESS	operation completed successfully.
GrB_PANIC	unknown internal error.
GrB_OUT_OF_MEMORY	not enough memory available for operation.
GrB_NULL_POINTER	at least one of utype, create, destroy, copy pointers is NULL.

Fig. 3. Definition of a GrB_Type_new GraphBLAS method that can support arbitrary user-defined types.

the current GraphBLAS API, but may be added in the future. Kronecker product may be useful for generating graphs. It is known that the Kronecker product of the adjacency matrices of two graphs is the adjacency matrix of their tensor product graph.

IX. Rank Promotion

Rank promotion is the conversion of an object of lower rank (*e.g.*, scalar or rank 0) to an object of a higher rank such as a vector (rank 1), or a matrix (rank 2). It is a common feature in array programming languages such as Fortran 90+ and MATLAB. Typically, a scalar is converted to a matrix or vector by replicating it in every element of the matrix or vector. A vector is converted to a matrix by replicating it either along the rows or along the columns of the matrix. The replication factor can be stated explicitly or implicitly calculated in order to result in a valid operation. We should note that rank promotion can already be accomplished explicitly with the existing GraphBLAS methods. However, doing it automatically, as we propose in this section, saves a matrix or vector instantiation just for that purpose.

In GraphBLAS, scalars, whether of built-in or user-defined type, are always of rank 0. Vectors and matrices are of rank 1 and 2, respectively. Our proposal is to support *automatic* rank promotion by allowing, in most cases, the use of an object of lower rank in an input argument.

For example, in the GraphBLAS matrix multiply method

GrB_mxm(C,Mask,accum,op,A,B,desc)

A and B are input matrices. One could replace either (or both) of them by a scalar or vector. Assume no transposition specified by the descriptor desc, and let C be an $m \times n$ matrix,

TABLE IV
Tentative list of automatic rank promotions in GraphBLAS.

Matrices: $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{M}$
Vectors: $\mathbf{a}, \mathbf{b}, \mathbf{u}, \mathbf{w}, \mathbf{m}$
Scalars: a, b, u, v
Δ denotes a descriptor
$\mathbb{S}$ is a semiring
$\odot$ is a binary operator used for accumulation
$\oplus$ and $\otimes$ are binary operators, either standalone or from a monoid/semiring
$f(\cdot)$ is a unary operator

Method	Promotions
GrB_mxm$(\mathbf{C}, \mathbf{M}, \odot, \mathbb{S}, \mathbf{A}, \mathbf{B}, \Delta)$	$a \to \mathbf{A}$ $\mathbf{a} \to \mathbf{A}$ $b \to \mathbf{B}$ $\mathbf{b} \to \mathbf{B}$
GrB_vxm$(\mathbf{w}, \mathbf{m}, \odot, \mathbb{S}, \mathbf{u}, \mathbf{A}, \Delta)$	$u \to \mathbf{u}$ $a \to \mathbf{A}$ $\mathbf{a} \to \mathbf{A}$
GrB_mxv$(\mathbf{w}, \mathbf{m}, \odot, \mathbb{S}, \mathbf{A}, \mathbf{u}, \Delta)$	$u \to \mathbf{u}$ $a \to \mathbf{A}$ $\mathbf{a} \to \mathbf{A}$
GrB_eWiseMult$(\mathbf{w}, \mathbf{m}, \odot, \otimes, \mathbf{u}, \mathbf{v}, \Delta)$	$u \to \mathbf{u}$ $v \to \mathbf{v}$
GrB_eWiseMult$(\mathbf{C}, \mathbf{M}, \odot, \otimes, \mathbf{A}, \mathbf{B}, \Delta)$	$a \to \mathbf{A}$ $\mathbf{a} \to \mathbf{A}$ $b \to \mathbf{B}$ $\mathbf{b} \to \mathbf{B}$
GrB_eWiseAdd$(\mathbf{w}, \mathbf{m}, \odot, \otimes, \mathbf{u}, \mathbf{v}, \Delta)$	$u \to \mathbf{u}$ $v \to \mathbf{v}$
GrB_eWiseAdd$(\mathbf{C}, \mathbf{M}, \odot, \otimes, \mathbf{A}, \mathbf{B}, \Delta)$	$a \to \mathbf{A}$ $\mathbf{a} \to \mathbf{A}$ $b \to \mathbf{B}$ $\mathbf{b} \to \mathbf{B}$
GrB_assign$(\mathbf{C}, \mathbf{M}, \odot, \mathbf{A}, \mathbf{i}, m, \mathbf{j}, n, \Delta)$	$a \to \mathbf{A}$ $\mathbf{a} \to \mathbf{A}$
GrB_apply$(\mathbf{w}, \mathbf{m}, \odot, f(\cdot), \mathbf{u}, \Delta)$	$u \to \mathbf{u}$
GrB_apply$(\mathbf{C}, \mathbf{M}, \odot, f(\cdot), \mathbf{A}, \Delta)$	$a \to \mathbf{A}$ $\mathbf{a} \to \mathbf{A}$

A be a scalar and B be an $p \times n$ matrix. The scalar A would be converted into an $m \times p$ matrix, with all its elements set to the value of A. After that, the matrix multiplication would proceed as specified in the standard. We note that the same requirements for domain compatibility would still hold.

If, instead, A were an m-element vector, it would be converted into an $m \times p$ matrix by replicating it p times as a column of the matrix. Replication as rows could be achieved by specifying transposition of A in the descriptor.

A tentative list of GraphBLAS methods supporting automatic rank promotion is shown in Table IV. The left column of the table shows the standard signatures of the main GraphBLAS methods. The right column shows which rank promotions are supported for that method. A promotion $a \to \mathbf{A}$ means that a scalar a can be used in place of matrix $\mathbf{A}$. That scalar will get promoted to a matrix of the right shape before the operation. Correspondingly, $\mathbf{a} \to \mathbf{A}$ means that a vector $\mathbf{a}$ can be used in place of matrix $\mathbf{A}$. Finally, $u \to \mathbf{u}$ means that a scalar u can be used in place of vector $\mathbf{u}$.

We realize that adding automatic rank promotion can result in operations that the user did not intend if an object with a the incorrect rank is accidentally sent as one of the input

arguments. This could, again, be ameliorated with the use of the descriptor by requiring flags corresponding to the input arguments that explicitly specify the type of promotion desired. Further discussion within the GraphBLAS community is desired before deciding on the approach.

X. Conclusion

The GraphBLAS C API 1.0 provisional specification was released in May of 2017. The qualifier "provisional" will be dropped once two conforming implementations of the specification have been completed. To manage the scope of the project, we had to defer many planned features for a future release of the GraphBLAS C API.

A successful API evolves over time to meet the needs of its user community. This means a dialogue between the users of the API and the team working on developing the API is critical. This paper is the start of the next phase in that dialog, launching an ongoing discussion of the future GraphBLAS C API version 2.0.

Acknowledgments and Disclaimers

We thank the members of the GraphBLAS forum.

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 [DM17-0410]. Aydin Buluc and Carl Yang were supported in part by the Applied Mathematics Program of the DOE Office of Advanced Scientific Computing Research under contract number DEAC02-05CH11231.

References

[1] The GraphBLAS Forum. http://graphblas.org/.

[2] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. Design of the GraphBLAS API for C. In *Intl. Parallel & Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 2017.

[3] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. The GraphBLAS C API Specification, version 1.0.0. Technical report, The GraphBLAS Signatures Subgroup, May 2017. http://graphblas.org/ aydin/GraphBLAS_API_C.pdf.

[4] 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.

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