The GraphBLAS C API Specification [†]:

Version 1.2.0

Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira, Carl Yang

Generated on 2018/05/18 at 12:14:46 EDT

[†]Based on *GraphBLAS Mathematics* by Jeremy Kepner

This version is a definitive release of the GraphBLAS C API specification. As of the date of this document, at least two independent and functionally complete implementations are available.

Contents

Acknowledgments . . . 7

1 Introduction **9**

2 Basic Concepts **11**

2.1 Glossary . . . 11

2.1.1 GraphBLAS API basic definitions . . . 11

2.1.2 GraphBLAS objects and their structure . . . 12

2.1.3 Algebraic structures used in the GraphBLAS . . . 13

2.1.4 The execution of an application using the GraphBLAS C API . . . 13

2.1.5 GraphBLAS methods: behaviors and error conditions . . . 14

2.2 Notation . . . 16

2.3 Algebraic and Arithmetic Foundations . . . 17

2.4 GraphBLAS Opaque Objects . . . 17

2.5 Domains . . . 19

2.6 Operators and Associated Functions . . . 19

2.7 Indices, Index Arrays, and Scalar Arrays . . . 21

2.8 Execution Model . . . 21

2.8.1 Execution modes . . . 22

2.8.2 Thread safety . . . 23

2.9 Error Model . . . 24

3 Objects **27**

3.1 Operators . . . 27

3.2 Monoids . . . 29

3.3 Semirings 29
3.4 Vectors 29
3.5 Matrices 30
3.6 Masks 31
3.7 Descriptors 31

4 Methods **35**
4.1 Context Methods 35
4.1.1 init: Initialize a GraphBLAS context 35
4.1.2 finalize: Finalize a GraphBLAS context 36
4.2 Object Methods 36
4.2.1 Algebra Methods 37
4.2.1.1 Type_new: Create a new GraphBLAS (user-defined) type 37
4.2.1.2 UnaryOp_new: Create a new GraphBLAS unary operator 38
4.2.1.3 BinaryOp_new: Create a new GraphBLAS binary operator 39
4.2.1.4 Monoid_new: Create new GraphBLAS monoid 40
4.2.1.5 Semiring_new: Create new GraphBLAS semiring 41
4.2.2 Vector Methods 42
4.2.2.1 Vector_new: Create new vector 42
4.2.2.2 Vector_dup: Create a copy of a GraphBLAS vector 43
4.2.2.3 Vector_clear: Clear a vector 44
4.2.2.4 Vector_size: Size of a vector 45
4.2.2.5 Vector_nvals: Number of stored elements in a vector 46
4.2.2.6 Vector_build: Store elements from tuples into a vector 47
4.2.2.7 Vector_setElement: Set a single element in a vector 49
4.2.2.8 Vector_extractElement: Extract a single element from a vector. . . . 50
4.2.2.9 Vector_extractTuples: Extract tuples from a vector 52
4.2.3 Matrix Methods 54
4.2.3.1 Matrix_new: Create new matrix 54
4.2.3.2 Matrix_dup: Create a copy of a GraphBLAS matrix 55
4.2.3.3 Matrix_clear: Clear a matrix 56

4.2.3.4 Matrix_nrows: Number of rows in a matrix 57
4.2.3.5 Matrix_ncols: Number of columns in a matrix 57
4.2.3.6 Matrix_nvals: Number of stored elements in a matrix 58
4.2.3.7 Matrix_build: Store elements from tuples into a matrix 59
4.2.3.8 Matrix_setElement: Set a single element in matrix 61
4.2.3.9 Matrix_extractElement: Extract a single element from a matrix . . . 62
4.2.3.10 Matrix_extractTuples: Extract tuples from a matrix 64
4.2.4 Descriptor Methods 66
4.2.4.1 Descriptor_new: Create new descriptor 66
4.2.4.2 Descriptor_set: Set content of descriptor 67
4.2.5 free method 68
4.3 GraphBLAS Operations 69
4.3.1 mxm: Matrix-matrix multiply 73
4.3.2 vxm: Vector-matrix multiply 77
4.3.3 mxv: Matrix-vector multiply 81
4.3.4 eWiseMult: Element-wise multiplication 86
4.3.4.1 eWiseMult: Vector variant 86
4.3.4.2 eWiseMult: Matrix variant 90
4.3.5 eWiseAdd: Element-wise addition 95
4.3.5.1 eWiseAdd: Vector variant 95
4.3.5.2 eWiseAdd: Matrix variant 100
4.3.6 extract: Selecting Sub-Graphs 105
4.3.6.1 extract: Standard vector variant 105
4.3.6.2 extract: Standard matrix variant 109
4.3.6.3 extract: Column (and row) variant 114
4.3.7 assign: Modifying Sub-Graphs 119
4.3.7.1 assign: Standard vector variant 119
4.3.7.2 assign: Standard matrix variant 124
4.3.7.3 assign: Column variant 130
4.3.7.4 assign: Row variant 135

4.3.7.5 assign: Constant vector variant 140
4.3.7.6 assign: Constant matrix variant 144
4.3.8 apply: Apply a unary function to the elements of an object 150
4.3.8.1 apply: Vector variant 150
4.3.8.2 apply: Matrix variant 154
4.3.9 reduce: Perform a reduction across the elements of an object 158
4.3.9.1 reduce: Standard matrix to vector variant 158
4.3.9.2 reduce: Vector-scalar variant 162
4.3.9.3 reduce: Matrix-scalar variant 165
4.3.10 transpose: Transpose rows and columns of a matrix 167
4.4 Sequence Termination 171
4.4.1 wait: Waits until pending operations complete 171
4.4.2 error: Get an error message regarding internal errors 172

5 Nonpolymorphic Interface **173**

A Revision History **179**

B Examples **181**
B.1 Example: breadth-first search (BFS) in GraphBLAS 182
B.2 Example: BFS in GraphBLAS using apply 183
B.3 Example: betweenness centrality (BC) in GraphBLAS 184
B.4 Example: batched BC in GraphBLAS 186
B.5 Example: maximal independent set (MIS) in GraphBLAS 188
B.6 Example: counting triangles in GraphBLAS 190

Acknowledgments

This document represents the work of the people who have served on the C API Subcommittee of the GraphBLAS Forum.

Those who served as C API Subcommittee members for GraphBLAS 1.0 and 1.1 are (in alphabetical order):

- Aydın Buluç (Lawrence Berkeley National Laboratory)
- Timothy G. Mattson (Intel Corporation)
- Scott McMillan (Software Engineering Institute at Carnegie Mellon University)
- José Moreira (IBM Corporation)
- Carl Yang (UC Davis)

The GraphBLAS 1.0 specification is based upon work funded and supported in part by:

- The Department of Energy Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231
- Intel Corporation
- Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute [DM-0003727]
- International Business Machines Corporation
- Department of Defense under contract No. W911QX-12-C-0059, L-3 Data Tactics subcontract SCT-14-004 with University of California, Davis

The following people provided valuable input and feedback during the development of the specification (in alphabetical order): Hollen Barmer, Tim Davis, Jeremy Kepner, Peter Kogge, Manoj Kumar, Andrew Mellinger, Maxim Naumov, Michael Wolf, Albert-Jan Yzelman.

Chapter 1

Introduction

The GraphBLAS standard defines a set of matrix and vector operations based on semi-ring algebraic structures. These operations can be used to express a wide range of graph algorithms. This document defines the C binding to the GraphBLAS standard. We refer to this as the *GraphBLAS C API* (Application Programming Interface).

The GraphBLAS C API is built on a collection of objects exposed to the C programmer as opaque data types. Functions that manipulate these objects are referred to as *methods*. These methods fully define the interface to GraphBLAS objects to create or destroy them, modify their contents, and copy the contents of opaque objects into non-opaque objects; the contents of which are under direct control of the programmer.

The GraphBLAS C API is designed to work with C99 (ISO/IEC 9899:199) extended with *static type-based* and *number of parameters-based* function polymorphism, and language extensions on par with the `_Generic` construct from C11 (ISO/IEC 9899:2011). Furthermore, the standard assumes programs using the GraphBLAS C API will execute on hardware that supports floating point arithmetic such as that defined by the IEEE 754 (IEEE 754-2008) standard.

The remainder of this document is organized as follows:

- Chapter 2: Basic Concepts
- Chapter 3: Objects
- Chapter 4: Methods
- Chapter 5: Nonpolymorphic Interface
- Appendix A: Revision History
- Appendix B: Examples

Chapter 2

Basic Concepts

The GraphBLAS C API is used to construct graph algorithms expressed "in the language of linear algebra". Graphs are expressed as matrices, and the operations over these matrices are generalized through the use of a semiring algebraic structure.

In this chapter, we will define the basic concepts used to define the GraphBLAS C API. We provide the following elements:

- Glossary of terms used in this document.
- Algebraic structures and associated arithmetic foundations of the API.
- Domains of elements in the GraphBLAS.
- Functions that appear in the GraphBLAS algebraic structures and how they are managed.
- Indices, index arrays, and scalar arrays used to expose the contents of GraphBLAS objects.
- The execution and error models implied by the GraphBLAS C specification.

2.1 Glossary

2.1.1 GraphBLAS API basic definitions

- *application*: A program that calls methods from the GraphBLAS C API to solve a problem.
- *GraphBLAS C API*: The application programming interface that fully defines the types, objects, literals, and other elements of the C binding to the GraphBLAS.
- *function*: Refers to a named group of statements in the C programming language. Methods, operators, and user-defined functions are typically implemented as C functions and when referring to the code programmers write, as opposed to the role of functions as an element of the GraphBLAS, they may be referred to as such.

- *method*: A function defined in the GraphBLAS C API that manipulates GraphBLAS objects or other opaque features of the implementation of the GraphBLAS API.
- *operator*: A function that performs an operation on the elements stored in GraphBLAS matrices and vectors.
- *GraphBLAS operation*: A mathematical operation defined in the GraphBLAS mathematical specification. These operations (not to be confused with *operators*) typically act on matrices and vectors with elements defined in terms of an algebraic semiring.

2.1.2 GraphBLAS objects and their structure

- *GraphBLAS object*: An instance of a data type defined by the GraphBLAS C API that is opaque and manipulated only through the API. There are three groups of GraphBLAS objects: *algebraic objects* (operators, monoids and semirings), *collections* (vectors, matrices and masks), and descriptors. Because the object is based on an opaque datatype, an implementation of the GraphBLAS C API has the flexibility to optimize data structures for a particular platform. GraphBLAS objects are often implemented as sparse data structures, meaning only the subset of the elements that have non-zero values are stored.
- *handle*: A variable that uses one of the GraphBLAS opaque data types. The value of this variable holds a reference to a GraphBLAS object but not the contents of object itself. Hence, assigning a value of one handle to another variable copies the reference to the GraphBLAS object but not the contents of the object.
- *non-opaque datatype*: Any datatype that exposes its internal structure. This is contrasted with an *opaque datatype* that hides its internal structure and can be manipulated only through an API.
- *domain*: The set of valid values for the elements of a GraphBLAS object. Note that some GraphBLAS objects involve functions that map values from one or more input domains onto values in an output domain. These GraphBLAS objects would have multiple domains.
- *structural zero*: Any element that has a valid index (or indices) in a GraphBLAS vector or matrix but is not explicitly identified in the list of elements of that vector or matrix. Also known as an *implied zero*. From a mathematical perspective, a *structural zero* is treated as having the value of the zero element of the relevant monoid or semiring.
- *mask*: An internal GraphBLAS object used to control how values are stored in a method's output object. The mask exists only inside a method; hence, it is called an *internal opaque object*. A mask is formed from the elements of a collection object (vector or matrix) input as a mask parameter to a method. An element of the mask exists for each element that exists in the input collection object when the value of that element cast to a Boolean type is `true`. Masks have structure but no values. That is, while a tuple for a vector or matrix has indices and values, tuples within a mask have indices but not values. Instead, we say that the tuples that exist within a mask have implied values of `true` while the structural zeros of the mask have implied values of `false`.

- *structural complement*: Operation on a mask where stored elements become *structural zeros* and vice versa. The *structural complement* of a GraphBLAS mask, M, is another mask, M', where the elements of M' are those elements from M that *do not* exist. In other words, elements of M with implied value `true` are `false` in M' while the structural zeros of M with implied values `false` are `true` in M'.

2.1.3 Algebraic structures used in the GraphBLAS

- *GraphBLAS operators*: Binary or unary operators that act on elements of GraphBLAS objects. *GraphBLAS operators* are used to express algebraic structures used in the GraphBLAS such as monoids and semirings. There are two types of *GraphBLAS operators*: (1) predefined operators found in Table 2.3 and (2) user-defined operators using GrB_UnaryOp_new() or GrB_BinaryOp_new() (see Section 4.2.1).

- *associative operator*: In an expression where a binary operator is used two or more times consecutively, that operator is *associative* if the result does not change regardless of the way operations are grouped (without changing their order) changes. In other words, in a sequence of binary operations using the same associative operator, the legal placement of parenthesis does not change the value resulting from the sequence operations. Operators that are associative over infinitely precise numbers (e.g., real numbers) are not strictly associative when applied to numbers with finite precision (e.g., floating point numbers). Such non-associativity results, for example, from roundoff errors or from the fact some numbers can not be represented exactly as floating point numbers. In the GraphBLAS specification, as is common practice in computing, we refer to operators as *associative* when their mathematical definition over infinitely precise numbers is associative even when they are only approximately associative when applied to finite precision numbers.

- *monoid*: An algebraic structure consisting of a domain, an associative binary operator, and an identity corresponding to that operator.

- *semiring*: An algebraic structure consisting of a set of allowed values (the *domain*), two commutative binary operators called addition and multiplication (where multiplication distributes over addition), and identities over addition (*0*) and multiplication (*1*). The additive identity is an annihilator over multiplication. Note that a *GraphBLAS semiring* is allowed to diverge from the mathematically rigorous definition of a semiring since certain combinations of domains, operators, and identity elements are useful in graph algorithms even when they do not strictly match the mathematical definition of a semiring.

2.1.4 The execution of an application using the GraphBLAS C API

- *program order*: The order of the GraphBLAS methods as defined by the text of an application program.

- *sequence*: A series of GraphBLAS method calls in program order. An implementation of the GraphBLAS may reorder or even fuse GraphBLAS methods within a sequence as long as the definitions of any GraphBLAS object that is later read by an application are not changed; by "read" we mean that values are copied from an opaque GraphBLAS object into a non-opaque object. A sequence begins when a thread calls the first method that creates or modifies a GraphBLAS object, either (1) the first call in an application or (2) the first call following termination of a prior sequence. The only way to terminate a sequence within an application is with a call to the GrB_wait() method.

- *complete*: The state of a GraphBLAS object when the computations that implement the mathematical definition of the object have finished and the values associated with that object are available to any method that would load them into a non-opaque data structure. A GraphBLAS object is fully defined by the sequence of methods. The execution of a sequence may be deferred, however, so at any point in an application, a GraphBLAS object may not be materialized; that is, the values associated with a particular GraphBLAS object may not have been computed and stored in memory. Essentially, methods that extract elements from an opaque object and copy them into a non-opaque object force completion of the opaque object.

- *materialize*: Cause the values associated with that object to be resident in memory and visible to an application. A GraphBLAS object has been *materialized* when the computations that implement the mathematical definition of the object are *complete*. A GraphBLAS object that is never loaded into a non-opaque data structure may potentially never be materialized. This might happen, for example, should the operations associated with the object be fused or otherwise changed by the runtime system that supports the implementation of the GraphBLAS C API.

- *context*: An instance of the GraphBLAS C API implementation as seen by an application. An application can have only one context between the start and end of the application. A context begins with the first thread that calls GrB_init() and ends with the first thread to call GrB_finalize(). It is an error for GrB_init() or GrB_finalize() to be called more than one time within an application. The context is used to constrain the behavior of an instance of the GraphBLAS C API implementation and support various execution strategies. Currently, the only supported constraints on a context pertain to the mode of program execution.

- *mode*: Defines how a GraphBLAS sequence executes, and is associated with the *context* of a GraphBLAS C API implementation. It is set by an application with its call to GrB_init() to one of two possible states. In *blocking mode*, GraphBLAS methods return after the computations complete and any output objects have been updated. In *nonblocking mode*, a method may return once the arguments are tested as consistent with the method (i.e., there are no API errors), and potentially before any computation has taken place.

2.1.5 GraphBLAS methods: behaviors and error conditions

- *implementation defined behavior*: Behavior that must be documented by the implementation and is allowed to vary among different compliant implementations.

- *undefined behavior*: Behavior that is not specified by the GraphBLAS C API. A conforming implementation is free to choose results delivered from a method whose behavior is undefined.

- *thread safe routine*: A routine that performs its intended function even when executed concurrently (by more than one thread).

- *shape compatible objects*: GraphBLAS objects (matrices and vectors) passed as parameters to a GraphBLAS method that have the correct number of dimensions and sizes for each dimension to satisfy the rules of the mathematical definition of the operation associated with the method. This is also referred to as *dimension compatible*.

- *domain compatible*: Two domains for which values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other, and a domain from a user-defined type is only compatible with itself. If any *domain compatibility* rule above is violated, execution of GraphBLAS method ends and the domain mismatch error GrB_DOMAIN_MISMATCH is returned.

2.2 Notation

Notation	Description
$D_{out}, D_{in}, D_{in_1}, D_{in_2}$	Refers to output and input domains of various GraphBLAS operators.
$\mathbf{D}_{out}(*), \mathbf{D}_{in}(*), \mathbf{D}_{in_1}(*), \mathbf{D}_{in_2}(*)$	Evaluates to output and input domains of GraphBLAS operators (usually a unary or binary operator, or semiring).
$\mathbf{D}(*)$	Evaluates to the (only) domain of a GraphBLAS object (usually a monoid, vector, or matrix).
f	An arbitrary unary function, usually a component of a unary operator.
$\mathbf{f}(F_u)$	Evaluates to the unary function contained in the unary operator given as the argument.
$\odot$	An arbitrary binary function, usually a component of a binary operator.
$\bigodot(*)$	Evaluates to the binary function contained in the binary operator or monoid given as the argument.
$\otimes$	Multiplicative binary operator of a semiring.
$\oplus$	Additive binary operator of a semiring.
$\bigotimes(S)$	Evaluates to the multiplicative binary operator of the semiring given as the argument.
$\bigoplus(S)$	Evaluates to the additive binary operator of the semiring given as the argument.
$\mathbf{0}(*)$	The identity of a monoid, or the additive identity of a GraphBLAS semiring.
$\mathbf{L}(*)$	The contents (all stored values) of the vector or matrix GraphBLAS objects. For a vector, it is the set of (index, value) pairs, and for a matrix it is the set of (row, col, value) triples.
$\mathbf{v}(i)$ or v_i	The i^{th} element of the vector $\mathbf{v}$.
$\mathbf{size}(\mathbf{v})$	The size of the vector $\mathbf{v}$.
$\mathbf{ind}(\mathbf{v})$	The set of indices corresponding to the stored values of the vector $\mathbf{v}$.
$\mathbf{nrows}(\mathbf{A})$	The number of rows in the $\mathbf{A}$.
$\mathbf{ncols}(\mathbf{A})$	The number of columns in the $\mathbf{A}$.
$\mathbf{indrow}(\mathbf{A})$	The set of row indices corresponding to rows in $\mathbf{A}$ that have stored values.
$\mathbf{indcol}(\mathbf{A})$	The set of column indices corresponding to columns in $\mathbf{A}$ that have stored values.
$\mathbf{ind}(\mathbf{A})$	The set of (i, j) indices corresponding to the stored values of the matrix.
$\mathbf{A}(i, j)$ or A_{ij}	The element of $\mathbf{A}$ with row index i and column index j.
$\mathbf{A}(:, j)$	The j^{th} column of the the matrix $\mathbf{A}$.
$\mathbf{A}(i, :)$	The i^{th} row of the the matrix $\mathbf{A}$.
$\mathbf{A}^T$	The transpose of the matrix $\mathbf{A}$.
$\neg\mathbf{M}$	The structural complement of $\mathbf{M}$.
$\widetilde{\mathbf{t}}$	A temporary object created by the GraphBLAS implementation.
$< type >$	A method argument type that is void * or one of the types from Table 2.2.
GrB_ALL	A method argument literal to indicate that all indices of an input array should be used.
GrB_Type	A method argument type that is either a user defined type or one of the types from Table 2.2.
GrB_Object	A method argument type referencing any of the GraphBLAS object types.
GrB_NULL	The GraphBLAS NULL.

2.3 Algebraic and Arithmetic Foundations

Graphs can be represented in terms of matrices. Operations defined by the GraphBLAS standard operate on these matrices to construct graph algorithms. These GraphBLAS operations are defined in terms of GraphBLAS semiring algebraic structures. Modifying the underlying semiring changes the result of an operation to support a wide range of graph algorithms.

Inside a given algorithm, it is often beneficial to change the GraphBLAS semiring that applies to an operation on a matrix. This has two implications on the C binding to the GraphBLAS. First, it means that we define a separate object for the semiring to pass into functions. Since in many cases the full semiring is not required, we also support passing monoids or even binary operators, which means the semiring is implied rather than explicitly stated.

Second, the ability to change semirings impacts the meaning of the *implied zero* in a sparse representation of a matrix. This element in real arithmetic is zero, which is the identity of the *addition* operator and the annihilator of the *multiplication* operator. As the semiring changes, this *implied* or *structural zero* changes to the identity of the *addition* operator and the annihilator of the *multiplication* operator for the new semiring. Nothing changes in the stored matrix, but the implied values within the sparse matrix change with respect to a particular operation. In most cases, the nature of the implied zero does not matter since the GraphBLAS treats these as elements of the matrix that do not exist. As we will see, however, there is a small subset of GraphBLAS methods (the element-wise operations) where to understand the method you need to understand the implied zero.

The mathematical formalism for graph operations in the language of linear algebra assumes that we can operate in the field of real numbers. However, the GraphBLAS C binding is designed for implementation on computers, which by necessity have a finite number of bits to represent numbers. Therefore, we require a conforming implementation to use floating point numbers such as those defined by the IEEE-754 standard (both single- and double-precision) wherever real numbers need to be represented. The practical implications of these finite precision numbers is that the result of a sequence of computations may vary from one execution to the next as the association of operations changes. While techniques are known to reduce these effects, we do not require or even expect an implementation to use them as they may add considerable overhead. The fact is that in most cases, these roundoff errors are not significant, and when they are significant, the problem itself is ill-conditioned and needs to be reformulated.

2.4 GraphBLAS Opaque Objects

Objects defined in the GraphBLAS standard include collections of elements (matrices and vectors), operators on those elements (unary and binary operators), and algebraic structures (semirings and monoids). GraphBLAS objects are defined as opaque types; that is, they are managed, manipulated, and accessed solely through the GraphBLAS application programming interface. This gives an implementation of the GraphBLAS C specification flexibility to optimize objects for different scenarios or to meet the needs of different hardware platforms.

A GraphBLAS opaque object is accessed through its handle. A handle is a variable that uses

Table 2.1: GraphBLAS opaque objects and their types.

GrB_Object types	Description
GrB_Type	User-defined scalar type.
GrB_UnaryOp	Unary operator, built-in or associated with a single-argument C function.
GrB_BinaryOp	Binary operator, built-in or associated with a two-argument C function.
GrB_Monoid	Monoid algebraic structure.
GrB_Semiring	A GraphBLAS semiring algebraic structure.
GrB_Matrix	Two-dimensional collection of elements; typically sparse.
GrB_Vector	One-dimensional collection of elements.
GrB_Descriptor	Descriptor object, used to modify behavior of methods.

one of the types from Table 2.1. An implementation of the GraphBLAS specification has a great deal of flexibility in how these handles are implemented. All that is required is that the handle corresponds to a type defined in the C language that supports assignment and comparison for equality. The GraphBLAS specification defines a literal GrB_INVALID_HANDLE that is valid for each type. Using the logical equality operator from C, it must be possible to compare a handle to GrB_INVALID_HANDLE to verify that a handle is valid.

An application using the GraphBLAS API will declare variables of the appropriate type for the objects it will use. Before use, the object must be initialized with the appropriate method. This is done with one of the methods that has a "_new" suffix in its name (e.g., GrB_Vector_new). Alternatively, an object can be initialized by duplicating an existing object with one of the methods that has the "_dup" suffix in its name (e.g., GrB_Vector_dup). When an application is finished with an object, any resources associated with that object can be released by a call to the GrB_free method.

These new, dup, and free methods are the only methods that change the value of a handle. Hence, objects changed by these methods are passed into the method as pointers. In all other cases, handles are not changed by the method and are passed by value. For example, even when multiplying matrices, while the contents of the output product matrix changes, the handle for that matrix is unchanged.

Programmers using GraphBLAS handles must be careful to distinguish between a handle and the object manipulated through a handle. For example, a program may declare two GraphBLAS objects of the same type, initialize one, and then assign it to the other variable. That assignment, however, only assigns the handle to the variable. It does not create a copy of that variable (to do that, one would need to use the appropriate duplication method). If later the object is freed by calling GrB_free with the first variable, the object is destroyed and the second variable is left referencing an object that no longer exists (a so called "dangling handle").

In addition to opaque objects manipulated through handles, the GraphBLAS C API defines an additional opaque object as an internal object; that is, the object is never exposed as a variable within an application. This opaque object is the mask used to control how computed values are stored in the output from a method. Masks are described in section 3.6.

Table 2.2: Predefined GrB_Type values, the corresponding C type (for scalar parameters), and domains for GraphBLAS.

GrB_Type values	C type	domain
GrB_BOOL	bool	$\{$false, true$\}$
GrB_INT8	int8_t	$\mathbb{Z} \cap [-2^7, 2^7)$
GrB_UINT8	uint8_t	$\mathbb{Z} \cap [0, 2^8)$
GrB_INT16	int16_t	$\mathbb{Z} \cap [-2^{15}, 2^{15})$
GrB_UINT16	uint16_t	$\mathbb{Z} \cap [0, 2^{16})$
GrB_INT32	int32_t	$\mathbb{Z} \cap [-2^{31}, 2^{31})$
GrB_UINT32	uint32_t	$\mathbb{Z} \cap [0, 2^{32})$
GrB_INT64	int64_t	$\mathbb{Z} \cap [-2^{63}, 2^{63})$
GrB_UINT64	uint64_t	$\mathbb{Z} \cap [0, 2^{64})$
GrB_FP32	float	IEEE 754 binary32
GrB_FP64	double	IEEE 754 binary64

2.5 Domains

GraphBLAS defines two kinds of collections: matrices and vectors. For any given collection, the elements of the collection belong to a *domain*, which is the set of valid values for the elements. In GraphBLAS, domains correspond to the valid values for types from the host language (in our case, the C programming language). For any variable or object V in GraphBLAS we denote as $\mathbf{D}(V)$ the domain of V, that is, the set of possible values that elements of V can take. The predefined types and corresponding domains used in the GraphBLAS are shown in Table 2.2. The Boolean type is defined in `stdbool.h`, the integral types are defined in `stdint.h`, and the floating point types are native to the language and in most cases defined by the IEEE-754 standard.

2.6 Operators and Associated Functions

GraphBLAS operators act on elements of GraphBLAS objects. A *binary operator* is a function that maps two input values to one output value. A *unary operator* is a function that maps one input value to one output value. The value of the output is determined by the value of the input(s). Binary operators are defined over two input domains and produce an output from a (possibly different) third domain. Unary operators are specified over one input domain and produce an output from a (possibly different) second domain.

Similar to GraphBLAS types with predefined types and user-defined types, GraphBLAS operators come in two types: (1) predefined operators found in Table 2.3 and (2) user-defined operators using GrB_UnaryOp_new() or GrB_BinaryOp_new() (see Section 4.2.1).

Table 2.3: Predefined unary and binary operators for GraphBLAS in C.

(a) Valid suffixes and corresponding C type (T in table (b)).

Suffix	C type
BOOL	`bool`
INT8	`int8_t`
UINT8	`uint8_t`
INT16	`int16_t`
UINT16	`uint16_t`
INT32	`int32_t`
UINT32	`uint32_t`
INT64	`int64_t`
UINT64	`uint64_t`
FP32	`float`
FP64	`double`

(b) Predefined Operators.

Operator type	GraphBLAS identifier	Domains	Description	
GrB_UnaryOp	GrB_IDENTITY_T	$T \to T$	$f(x) = x,$	identity
GrB_UnaryOp	GrB_AINV_T	$T \to T$	$f(x) = -x,$	additive inverse
GrB_UnaryOp	GrB_MINV_T	$T \to T$	$f(x) = \frac{1}{x},$	multiplicative inverse
GrB_UnaryOp	GrB_LNOT	`bool` $\to$ `bool`	$f(x) = \neg x,$	logical inverse
GrB_BinaryOp	GrB_LOR	`bool` $\times$ `bool` $\to$ `bool`	$f(x, y) = x \vee y,$	logical OR
GrB_BinaryOp	GrB_LAND	`bool` $\times$ `bool` $\to$ `bool`	$f(x, y) = x \wedge y,$	logical AND
GrB_BinaryOp	GrB_LXOR	`bool` $\times$ `bool` $\to$ `bool`	$f(x, y) = x \oplus y,$	logical XOR
GrB_BinaryOp	GrB_EQ_T	$T \times T \to$ `bool`	$f(x, y) = (x == y)$	equal
GrB_BinaryOp	GrB_NE_T	$T \times T \to$ `bool`	$f(x, y) = (x \neq y)$	not equal
GrB_BinaryOp	GrB_GT_T	$T \times T \to$ `bool`	$f(x, y) = (x > y)$	greater than
GrB_BinaryOp	GrB_LT_T	$T \times T \to$ `bool`	$f(x, y) = (x < y)$	less than
GrB_BinaryOp	GrB_GE_T	$T \times T \to$ `bool`	$f(x, y) = (x \geq y)$	greater than or equal
GrB_BinaryOp	GrB_LE_T	$T \times T \to$ `bool`	$f(x, y) = (x \leq y)$	less than or equal
GrB_BinaryOp	GrB_FIRST_T	$T \times T \to T$	$f(x, y) = x,$	first argument
GrB_BinaryOp	GrB_SECOND_T	$T \times T \to T$	$f(x, y) = y,$	second argument
GrB_BinaryOp	GrB_MIN_T	$T \times T \to T$	$f(x, y) = (x < y)\ ?\ x : y,$	minimum
GrB_BinaryOp	GrB_MAX_T	$T \times T \to T$	$f(x, y) = (x > y)\ ?\ x : y,$	maximum
GrB_BinaryOp	GrB_PLUS_T	$T \times T \to T$	$f(x, y) = x + y,$	addition
GrB_BinaryOp	GrB_MINUS_T	$T \times T \to T$	$f(x, y) = x - y,$	subtraction
GrB_BinaryOp	GrB_TIMES_T	$T \times T \to T$	$f(x, y) = xy,$	multiplication
GrB_BinaryOp	GrB_DIV_T	$T \times T \to T$	$f(x, y) = \frac{x}{y},$	division

2.7 Indices, Index Arrays, and Scalar Arrays

In order to interface with third-party software (i.e., software other than an implementation of the GraphBLAS), operations such as GrB_Matrix_build (§ 4.2.3.7) and GrB_Matrix_extractTuples (§ 4.2.3.10) must specify how the data should be laid out in non-opaque data structures. To this end we explicitly define the types for indices and the arrays used by these operations.

For indices a typedef is used to give a GraphBLAS name to a concrete type. We define it as follows:

```
typedef uint64_t GrB_Index;
```

An index array is a pointer to a set of GrB_Index values that are stored in a contiguous block of memory (i.e., GrB_Index*). Likewise, a scalar array is a pointer to a contiguous block of memory storing a number of scalar values as specified by the user. Some GraphBLAS operations (e.g., GrB_assign) include an input parameter with the type of an index array. This input index array selects a subset of elements from a GraphBLAS vector object to be used in the operation. In these cases, the literal GrB_ALL can be used in place of the index array input parameter to indicate that all indices of the associated GraphBLAS vector object should be used. As with any literal defined in the GraphBLAS, an implementation of the GraphBLAS C API has considerable freedom in terms of how GrB_ALL is defined. Since it is used as an argument for an array parameter, GrB_ALL must use a type consistent with a pointer, and it must have a non-null value so it can be distinguished from the erroneous case of passing a NULL pointer as an array.

2.8 Execution Model

A program using the GraphBLAS C API constructs GraphBLAS objects, manipulates them to implement a graph algorithm, and then extracts values from the GraphBLAS objects as the result of the algorithm. Functions defined within the GraphBLAS C API that manipulate GraphBLAS objects are called *methods*. If the method corresponds to one of the operations defined in the GraphBLAS mathematical specification, we refer to the method as an *operation*.

Graph algorithms are expressed as an ordered collection of GraphBLAS method calls defined by the order they are encountered in a program. This is called the *program order*. Each method in the collection uniquely and unambiguously defines the output GraphBLAS objects based on the GraphBLAS operation and the input GraphBLAS objects. This is the case as long as there are no execution errors, which can put objects in an invalid state (see § 2.9).

The GraphBLAS method calls in program order are organized into contiguous and nonoverlapping *sequences*. A sequence is an ordered collection of method calls as encountered by an executing thread. (For more on threads and GraphBLAS, see § 2.8.2.) A sequence begins with either (i) the first GraphBLAS method called by a thread, or (ii) the first method called by a thread after the end of the previous sequence. A sequence always ends (terminates) with a call to the GraphBLAS GrB_wait() method.

The GraphBLAS objects are fully defined at any point in a sequence by the methods in the sequence as long as there are no execution errors. In particular, as soon as a GraphBLAS method call returns, its output can be used in the next GraphBLAS method call. However, individual operations in a sequence may not be *complete*. We say that an operation is complete when all the computations in the operation have finished and all the values of its output object have been produced and committed to the address space of the program.

The opaqueness of GraphBLAS objects allows execution to proceed from one method to the next even when operations are not complete. Processing of nonopaque objects is never deferred in GraphBLAS. That is, methods that consume nonopaque objects (e.g., GrB_Matrix_build, § 4.2.3.7()) and methods that produce nonopaque objects (e.g., GrB_Matrix_extractTuples(), § 4.2.3.10) always finish consuming or producing those nonopaque objects before returning. Furthermore, methods that extract values from opaque GraphBLAS objects into nonopaque user objects (see Table 2.4) always force completion of all pending computations on the corresponding GraphBLAS source object.

Table 2.4: Methods that extract values from a GraphBLAS object, thereby forcing completion of the operations contributing to that particular object.

Method	Section
GrB_Vector_nvals	4.2.2.5
GrB_Vector_extractElement	4.2.2.8
GrB_Vector_extractTuples	4.2.2.9
GrB_Matrix_nvals	4.2.3.6
GrB_Matrix_extractElement	4.2.3.9
GrB_Matrix_extractTuples	4.2.3.10
GrB_reduce (vector-scalar variant)	4.3.9.2
GrB_reduce (matrix-scalar variant)	4.3.9.3

2.8.1 Execution modes

The execution model implied by GraphBLAS sequences depends on the *execution mode* of the GraphBLAS program. There are two modes: *blocking* and *nonblocking*.

- *blocking*: In blocking mode, each method completes the GraphBLAS operation defined by the method before proceeding to the next statement in program order. Output GraphBLAS objects defined by a method are fully produced and stored in memory (i.e., they are *materialized*). In other words, it is as if each method call is its own sequence. Even mechanisms that break the opaqueness of the GraphBLAS objects (e.g., performance monitors, debuggers, memory dumps) will observe the operation as complete.

- *nonblocking*: In nonblocking mode, each method may return once the input arguments have been inspected and verified to define a well formed GraphBLAS operation. (That is, there are no API errors. See § 2.9.) The GraphBLAS operation may not have completed, but the output object is ready to be used by the next GraphBLAS method call. Completion of *all* operations in a sequence, including any that may generate execution errors, is guaranteed once the sequence terminates. Sequence termination is accomplished by a call to GrB_wait().

An application executing in nonblocking mode is not required to return immediately after input arguments have been verified. A conforming implementation of the GraphBLAS C API running in nonblocking mode may choose to execute *as if* in blocking mode. Further, a sequence in nonblocking mode where every GraphBLAS operation is followed by a GrB_wait() call is equivalent to the same sequence in blocking mode with GrB_wait() calls removed.

Nonblocking mode allows for any execution strategy that satisfies the mathematical definition of the sequence. The methods can be placed into a queue and deferred. They can be chained together and fused (e.g., replacing a chained pair of matrix products with a matrix triple product). Lazy evaluation, greedy evaluation, and asynchronous execution are all valid as long as the final result agrees with the mathematical definition provided by the sequence of GraphBLAS method calls appearing in program order.

Blocking mode forces an implementation to carry out precisely the GraphBLAS operations defined by the methods and to store output objects to memory between method calls. It is valuable for debugging or in cases where an external tool such as a debugger needs to evaluate the state of memory during a sequence.

In a mathematically well-defined sequence with input objects that are well-conditioned and free of execution errors, the results from blocking and nonblocking modes should be identical outside of effects due to roundoff errors associated with floating point arithmetic. Due to the great flexibility afforded to an implementation when using nonblocking mode, we expect execution of a sequence in nonblocking mode to potentially complete execution in less time.

The mode is defined in the GraphBLAS C API when the context of the library invocation is defined. This occurs once before any GraphBLAS methods are called with a call to the GrB_init() function. This function takes a single argument of type GrB_Mode with the following possible values:

- GrB_BLOCKING Specifies the blocking mode context.
- GrB_NONBLOCKING Specifies the blocking mode context.

After all GraphBLAS methods are complete, the context is terminated with a call to GrB_finalize(). In the current version of the GraphBLAS C API, the context can be set only once in the execution of a program. That is, after GrB_finalize() is called, a subsequent call to GrB_init() is not allowed.

2.8.2 Thread safety

The GraphBLAS C API is designed to work in applications that execute with multiple threads; however, management of threads is not exposed within the definition of the GraphBLAS C API. The mapping of GraphBLAS methods onto threads and explicit synchronization between methods running on different threads are not defined. Furthermore, errors exposed within the error model (see section 2.9) are not required to manage information at a per-thread granularity.

The only requirement concerning the needs of multi-threaded execution found in the GraphBLAS C API is that implementations of GraphBLAS methods must be thread safe. Different threads may create GraphBLAS sequences that do not conflict and expect the results to be the same (within floating point roundoff errors) regardless of whether the sequences execute serially or concurrently.

Sequences that do not conflict are free of data races. A data race occurs when (1) two or more threads access shared objects, (2) those access operations include at least one modify operation, and (3) those operations are not ordered through synchronization operations. The GraphBLAS C API does not provide synchronization operations to define ordered accesses to GraphBLAS objects. Hence the only way to assure that two sequences running concurrently on different threads do not conflict is if neither sequence writes to an object that the other sequence either reads or writes.

2.9 Error Model

All GraphBLAS methods return a value of type GrB_Info to provide information available to the system at the time the method returns. The returned value can be either GrB_SUCCESS or one of the defined error values shown in Table 2.5. The errors fall into two groups: API errors (Table 2.5(a)) and execution errors (Table 2.5(b)).

An API error means a GraphBLAS method was called with parameters that violate the rules for that method. These errors are restricted to those that can be determined by inspecting the types and domains of GraphBLAS objects, GraphBLAS operators, or the values of scalar parameters fixed at the time a method is called. API errors are deterministic and consistent across platforms and implementations. API errors are never deferred, even in nonblocking mode. That is, if a method is called in a manner that would generate an API error, it always returns with the appropriate API error value. If a GraphBLAS method returns with an API error, it is guaranteed that none of the arguments to the method (or any other program data) have been modified.

Execution errors indicate that something went wrong during the execution of a legal GraphBLAS method invocation. Their occurrence may depend on specifics of the executing environment and data values being manipulated. This does not mean that execution errors are the fault of the GraphBLAS implementation. For example, a memory leak could arise from an error in an application's source code (a "program error"), but it may manifest itself in different points of a program's execution (or not at all) depending on the platform, problem size, or what else is running at that time. Index-out-of-bounds and insuficient space execution errors always indicate a program error.

In blocking mode, where each method executes to completion, a returned execution error value applies to the specific method. If a GraphBLAS method, executing in blocking mode, returns with any execution error from Table 2.5(b) other than GrB_PANIC, it is guaranteed that no argument used as input-only has been modified. Output arguments may be left in an invalid state, and their use downstream in the program flow may cause additional errors. If a GraphBLAS method returns with a GrB_PANIC execution error, no guarantees can be made about the state of any program data.

In nonblocking mode, execution errors can be deferred. A return value of GrB_SUCCESS only guarantees that there are no API errors in the method invocation. If an execution error value is returned by a method in nonblocking mode, it indicates that an error was found during execution of the sequence, up to and including the GrB_wait() method call that ends the sequence. When possible, that return value will provide information concerning the cause of the error.

If a GraphBLAS method, executing in nonblocking mode, returns with any execution error from Table 2.5(b) other than GrB_PANIC, it is guaranteed that no argument used as input-only through

```
const char *GrB_error();
```

Figure 2.1: Signature of GrB_error() function.

the entire sequence has been modified. Any output argument in the sequence may be left in an invalid state and its use downstream in the program flow may cause additional errors. If a GraphBLAS method returns with a GrB_PANIC, no guarantees can be made about the state of any program data.

After a call to any GraphBLAS method, the program can retrieve additional error information (beyond the error code returned by the method) though a call to the function GrB_error(). The signature of that function is shown in Figure 2.1. The function returns a pointer to a NULL-terminated string, and the contents of that string are implementation dependent. In particular, a null string (not a NULL pointer) is always a valid error string. The pointer is valid until the next call to any GraphBLAS method by the same thread. GrB_error() is a thread-safe function, in the sense that multiple threads can call it simultaneously and each will get its own error string back, referring to the last GraphBLAS method it called.

Table 2.5: Error values returned by GraphBLAS methods.

(a) API errors

Error code	Description
GrB_UNINITIALIZED_OBJECT	A GraphBLAS object is passed to a method before new was called on it.
GrB_NULL_POINTER	A NULL is passed for a pointer parameter.
GrB_INVALID_VALUE	Miscellaneous incorrect values.
GrB_INVALID_INDEX	Indices passed are larger than dimensions of the matrix or vector being accessed.
GrB_DOMAIN_MISMATCH	A mismatch between domains of collections and operations when user-defined domains are in use.
GrB_DIMENSION_MISMATCH	Operations on matrices and vectors with incompatible dimensions.
GrB_OUTPUT_NOT_EMPTY	An attempt was made to build a matrix or vector using an output object that already contains valid tuples (elements).
GrB_NO_VALUE	A location in a matrix or vector is being accessed that has no stored value at the specified location.

(b) Execution errors

Error code	Description
GrB_OUT_OF_MEMORY	Not enough memory for operations.
GrB_INSUFFICIENT_SPACE	The array provided is not large enough to hold output.
GrB_INVALID_OBJECT	One of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error.
GrB_INDEX_OUT_OF_BOUNDS	Reference to a vector or matrix element that is outside the defined dimensions of the object.
GrB_PANIC	Unknown internal error.

Chapter 3

Objects

The GraphBLAS *algebraic objects* operators, monoids, and semirings are presented below. These objects can be used as input arguments to various GraphBLAS operations, as shown in Table 3.1. The specific rules for each algebraic object are explained in the respective sections of those objects. A summary of the properties and recipes for building these GraphBLAS algebraic objects is presented in Table 3.2.

Once algebraic objects (operators, monoids, and semirings) are described, we introduce *collections* (vectors, matrices, and masks) that algebraic objects operate on. Finally, we introduce *descriptors*, which are a simple way to modify how algebraic objects operate on collections. More concretely, descriptors can be used (among other things) to perform multiplication with transpose of matrix without the user having to manually transpose the collection. A complete list of what descriptors are capable of can be found in the section.

3.1 Operators

A GraphBLAS *binary operator* $F_b = \langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$ is defined by three domains, D_{out}, D_{in_1}, D_{in_2}, and an operation $\odot : D_{in_1} \times D_{in_2} \rightarrow D_{out}$. For a given GraphBLAS operator $F_b = \langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$, we define $\mathbf{D}_{out}(F_b) = D_{out}$, $\mathbf{D}_{in_1}(F_b) = D_{in_1}$, $\mathbf{D}_{in_2}(F_b) = D_{in_2}$, and $\bigodot(F_b) = \odot$. Note that $\odot$ could be used in place of either $\oplus$ or $\otimes$ in other methods and operations.

A GraphBLAS *unary operator* $F_u = \langle D_{out}, D_{in}, f \rangle$ is defined by two domains, D_{out} and D_{in}, and an operation $f : D_{in} \rightarrow D_{out}$. For a given GraphBLAS operator $F_u = \langle D_{out}, D_{in}, f \rangle$, we define $\mathbf{D}_{out}(F_u) = D_{out}$, $\mathbf{D}_{in}(F_u) = D_{in}$, and $\mathbf{f}(F_u) = f$.

Table 3.1: Operator input for relevant GraphBLAS operations. The semiring add and times are shown if applicable.

Operation	Operator input
mxm, mxv, vxm	semiring
eWiseAdd	binary operator monoid semiring
eWiseMult	binary operator monoid semiring
reduce (to vector)	binary operator monoid
reduce (to scalar)	monoid
apply	unary operator
dup argument (build methods)	binary operator
accum argument (various methods)	binary operator

Table 3.2: Properties and recipes for building GraphBLAS algebraic objects: unary operator, binary operator, monoid, and semiring (composed of operations *add* and *times*).
Note 1: The output domain of the semiring times must be same as the domain of the semiring add. This ensures three domains for a semiring rather than four.

(a) Properties of algebraic objects.

Object	Must be commutative	Must be associative	Identity must exist	Number of domains
Unary operator	no	no	no	2
Binary operator	no	no	no	3
Monoid	no	yes	yes	1
Semiring add	yes	yes	yes	1
Semiring times	no	no	no	3 (see Note 1)

(b) Recipes for algebraic objects.

Object	Recipe	Number of domains
Unary operator	Function pointer	2
Binary operator	Function pointer	3
Monoid	Associative binary operator with identity	1
Semiring	Commutative monoid + binary operator	3

3.2 Monoids

A GraphBLAS *monoid* (or *monoid* for short) $M = \langle D, \odot, 0\rangle$ is defined by a single domain D, an *associative* [1] operation $\odot : D \times D \to D$, and an identity element $0 \in D$. For a given GraphBLAS monoid $M = \langle D, \odot, 0\rangle$ we define $\mathbf{D}(M) = D$, $\bigodot(M) = \odot$, and $\mathbf{0}(M) = 0$. A GraphBLAS monoid is equivalent to the conventional *monoid* algebraic structure.

Let $F = \langle D, D, D, \odot\rangle$ be an associative GraphBLAS binary operator with identity element $0 \in D$. Then $M = \langle F, 0\rangle = \langle D, \odot, 0\rangle$ is a GraphBLAS monoid. If $\odot$ is commutative, then M is said to be a *commutative monoid*. If a monoid M is created using an operator $\odot$ that is not associative, the outcome of GraphBLAS operations using such a monoid is undefined.

3.3 Semirings

A GraphBLAS *semiring* (or *semiring* for short) $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0\rangle$ is defined by three domains D_{out}, D_{in_1}, and D_{in_2}; an *associative* [2] and commutative additive operation $\oplus : D_{out} \times D_{out} \to D_{out}$; a multiplicative operation $\otimes : D_{in_1} \times D_{in_2} \to D_{out}$; and an identity element $0 \in D_{out}$. For a given GraphBLAS semiring $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0\rangle$ we define $\mathbf{D}_{in_1}(S) = D_{in_1}$, $\mathbf{D}_{in_2}(S) = D_{in_2}$, $\mathbf{D}_{out}(S) = D_{out}$, $\bigoplus(S) = \oplus$, $\bigotimes(S) = \otimes$, and $\mathbf{0}(S) = 0$.

Let $F = \langle D_{out}, D_{in_1}, D_{in_2}, \otimes\rangle$ be an operator and let $A = \langle D_{out}, \oplus, 0\rangle$ be a commutative monoid, then $S = \langle A, F\rangle = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0\rangle$ is a semiring.

Note: There must be one GraphBLAS monoid in every semiring which serves as the semiring's additive operator and specifies the same domain for its inputs and output parameters. If this monoid is not a commutative monoid, the outcome of GraphBLAS operations using the semiring is undefined.

A UML diagram of the conceptual hierarchy of object classes in GraphBLAS algebra (binary operators, monoids, and semirings) is shown in Figure 3.1.

3.4 Vectors

A vector $\mathbf{v} = \langle D, N, \{(i, v_i)\}\rangle$ is defined by a domain D, a size $N > 0$, and a set of tuples (i, v_i) where $0 \le i < N$ and $v_i \in D$. A particular value of i can appear at most once in $\mathbf{v}$. We define $\mathbf{size}(\mathbf{v}) = N$ and $\mathbf{L}(\mathbf{v}) = \{(i, v_i)\}$. The set $\mathbf{L}(\mathbf{v})$ is called the *content* of vector $\mathbf{v}$. We also define the set $\mathbf{ind}(\mathbf{v}) = \{i : (i, v_i) \in \mathbf{L}(\mathbf{v})\}$ (called the *structure* of $\mathbf{v}$), and $\mathbf{D}(\mathbf{v}) = D$. For a vector $\mathbf{v}$, $\mathbf{v}(i)$ is a reference to v_i if $(i, v_i) \in \mathbf{L}(\mathbf{v})$ and is undefined otherwise.

[1] It is expected that implementations of the GraphBLAS will utilize floating point arithmetic such as that defined in the IEEE-754 standard even though floating point arithmetic is not strictly associative.

[2] It is expected that implementations of the GraphBLAS will utilize floating point arithmetic such as that defined in the IEEE-754 standard even though floating point arithmetic is not strictly associative.

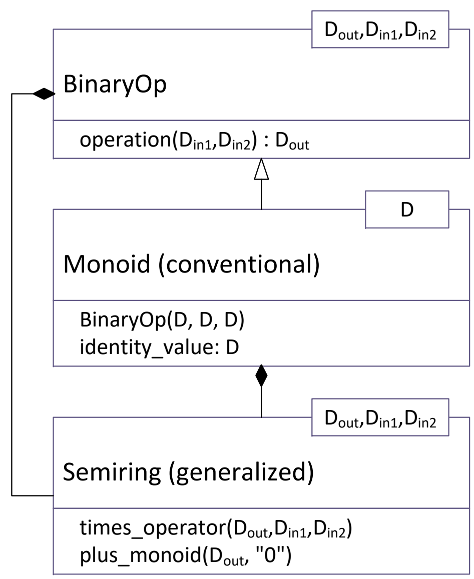

Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

3.5 Matrices

A matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\}\rangle$ is defined by a domain D, its number of rows $M > 0$, its number of columns $N > 0$, and a set of tuples (i, j, A_{ij}) where $0 \le i < M$, $0 \le j < N$, and $A_{ij} \in D$. A particular pair of values i, j can appear at most once in $\mathbf{A}$. We define $\mathbf{ncols}(\mathbf{A}) = N$, $\mathbf{nrows}(\mathbf{A}) = M$, and $\mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\}$. The set $\mathbf{L}(\mathbf{A})$ is called the *content* of matrix $\mathbf{A}$. We also define the sets $\mathbf{indrow}(\mathbf{A}) = \{i : \exists(i, j, A_{ij}) \in \mathbf{A}\}$ and $\mathbf{indcol}(\mathbf{A}) = \{j : \exists(i, j, A_{ij}) \in \mathbf{A}\}$. (These are the sets of nonempty rows and columns of $\mathbf{A}$, respectively.) The *structure* of matrix $\mathbf{A}$ is the set $\mathbf{ind}(\mathbf{A}) = \{(i, j) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}$, and $\mathbf{D}(\mathbf{A}) = D$. For a matrix $\mathbf{A}$, $\mathbf{A}(i, j)$ is a reference to A_{ij} if $(i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})$ and is undefined otherwise.

If $\mathbf{A}$ is a matrix and $0 \le j < N$, then $\mathbf{A}(:, j) = \langle D, M, \{(i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle$ is a vector called the j-th *column* of $\mathbf{A}$. Correspondingly, if $\mathbf{A}$ is a matrix and $0 \le i < M$, then $\mathbf{A}(i, :) = \langle D, N, \{(j, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle$ is a vector called the i-th *row* of $\mathbf{A}$.

Given a matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\}\rangle$, its *transpose* is another matrix $\mathbf{A}^T = \langle D, N, M, \{(j, i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle$.

3.6 Masks

The GraphBLAS C API defines an opaque object called a *mask*. The mask is used to control how computed values are stored in the output from a method. The mask is an *internal* opaque object; that is, it is never exposed as a variable within an application.

The mask is formed from objects input to the method that uses the mask. For example, a GraphBLAS method may be called with a matrix as the mask parameter. The internal mask object is constructed from the input matrix with an element of the mask for each tuple that exists in the matrix for which the value of the tuple cast to Boolean is *true*.

The internal mask object can be either a one- or a two-dimensional construct. One- and two-dimensional masks, described more formally below, are similar to vectors and matrices, respectively, except that they have structure (indices) but no values. When needed, a value is implied for the elements of a mask with an implied value of `true` for elements that exist and an implied value of `false` for elements that do not exist (i.e., the structural zeros of the mask imply a value of `false`). Hence, even though a mask does not contain any values, it can be considered to imply values from a Boolean domain.

A one-dimensional mask $\mathbf{m} = \langle N, \{i\}\rangle$ is defined by its number of elements $N > 0$, and a set $\mathbf{ind}(\mathbf{m})$ of indices $\{i\}$ where $0 \leq i < N$. A particular value of i can appear at most once in $\mathbf{m}$. We define $\mathbf{size}(\mathbf{m}) = N$. The set $\mathbf{ind}(\mathbf{m})$ is called the *structure* of mask $\mathbf{m}$.

A two-dimensional mask $\mathbf{M} = \langle M, N, \{(i,j)\}\rangle$ is defined by its number of rows $M > 0$, its number of columns $N > 0$, and a set $\mathbf{ind}(\mathbf{M})$ of tuples (i,j) where $0 \leq i < M$, $0 \leq j < N$. A particular pair of values i, j can appear at most once in $\mathbf{M}$. We define $\mathbf{ncols}(\mathbf{M}) = N$, and $\mathbf{nrows}(\mathbf{M}) = M$. We also define the sets $\mathbf{indrow}(\mathbf{M}) = \{i : \exists (i,j) \in \mathbf{ind}(\mathbf{M})\}$ and $\mathbf{indcol}(\mathbf{M}) = \{j : \exists (i,j) \in \mathbf{ind}(\mathbf{M})\}$. These are the sets of nonempty rows and columns of $\mathbf{M}$, respectively. The set $\mathbf{ind}(\mathbf{M})$ is called the *structure* of mask $\mathbf{M}$.

One common operation on masks is the *structural complement*. For a one-dimensional mask $\mathbf{m}$ this is denoted as $\neg\mathbf{m}$. For a two-dimensional masks, this is denoted as $\neg\mathbf{M}$. The structure of the complement of a one-dimensional mask $\mathbf{m}$ is defined as $\mathbf{ind}(\neg\mathbf{m}) = \{i : 0 \leq i < N, i \notin \mathbf{ind}(\mathbf{m})\}$. It is the set of all possible indices that do not appear in $\mathbf{m}$. The structure of the complement of a two-dimensional mask $\mathbf{M}$ is defined as the set $\mathbf{ind}(\neg\mathbf{M}) = \{(i,j) : 0 \leq i < M, 0 \leq j < N, (i,j) \notin \mathbf{ind}(\mathbf{M})\}$. It is the set of all possible indices that do not appear in $\mathbf{M}$.

3.7 Descriptors

Descriptors are used to modify the behavior of a GraphBLAS method. When present in the signature of a method, they appear as the last argument in the method. Descriptors specify how the other input arguments corresponding to GraphBLAS collections – vectors, matrices, and masks – should be processed (modified) before the main operation of a method is performed.

The descriptor is a lightweight object. It is composed of (field, value) pairs where the *field* selects one of the GraphBLAS objects from the argument list of a method and the *value* defines the indicated modification associated with that object. For example, a descriptor may specify that a particular

Table 3.3: Descriptors are GraphBLAS objects passed as arguments to Graph_BLAS operations to modify other GraphBLAS objects in the operation's argument list. A descriptor, desc, has one or more (field, value) pairs indicated as desc[GrB_Desc_Field].GrB_Desc_Value. In this table, we define all types and literals used with descriptors.

(a) Types used with GraphBLAS descriptors.

Type	Description
GrB_Descriptor	Type of a GraphBLAS descriptor object.
GrB_Desc_Field	Type of a descriptor field.
GrB_Desc_Value	Type of a descriptor field's value.

(b) Descriptor field names of type GrB_Desc_Field.

Field name	Description
GrB_OUTP	Field name for the output GraphBLAS object.
GrB_INP0	Field name for the first input GraphBLAS object.
GrB_INP1	Field name for the second input GraphBLAS object.
GrB_MASK	Field name for the mask GraphBLAS object.

(c) Descriptor field values of type GrB_Desc_Value.

Field Value	Description
GrB_SCMP	Use the structural complement of the associated object.
GrB_TRAN	Use the transpose of the associated object.
GrB_REPLACE	Clear the output object before assigning computed values.

input matrix needs to be transposed or that a mask needs to be structurally complemented (defined in Section 3.6) before using it in the operation.

For the purpose of constructing descriptors, the arguments of a method that can be modified are identified by specific field names. The output parameter (typically the first parameter in a GraphBLAS method) is indicated by the field name, GrB_OUTP. The mask is indicated by the GrB_MASK field name. The input parameters corresponding to the input vectors and matrices are indicated by GrB_INP0 and GrB_INP1 in the order they appear in the signature of the GraphBLAS method. The descriptor is an opaque object and hence we do not define how objects of this type should be implemented. When referring to (field, value) pairs for a descriptor, however, we often use the informal notation desc[GrB_Desc_Field].GrB_Desc_Value (without implying that a descriptor is to be implemented as an array of structures). We summarize all types, field names, and values used with descriptors in Table 3.3.

In the definitions of the GraphBLAS methods, we often refer to the *default behavior* of a method with respect to the action of a descriptor. If a descriptor is not provided or if the value associated with a particular field in a descriptor is not set, the default behavior of a GraphBLAS method is defined as follows:

- Input matrices are not transposed.
- The mask is used as is, without a structural complement.
- Values of the output object that are not directly modified by the operation are preserved.

Chapter 4

Methods

This chapter defines the behavior of all the methods in the GraphBLAS C API. All methods can be declared for use in programs by including the `GraphBLAS.h` header file.

4.1 Context Methods

The methods in this section set up and tear down the GraphBLAS context within which all GraphBLAS methods must be executed. The initialization of this context also includes the specification of which execution mode is to be used.

4.1.1 init: Initialize a GraphBLAS context

Creates and initializes a GraphBLAS C API context.

C Syntax

```
GrB_Info GrB_init(GrB_Mode mode);
```

Parameters

mode Mode for the GraphBLAS context. Must be either GrB_BLOCKING or GrB_NONBLOCKING.

Return Values

GrB_SUCCESS operation completed successfully.

GrB_PANIC unknown internal error.

GrB_INVALID_VALUE invalid mode specified, or method called multiple times.

Description

Creates and initializes a GraphBLAS C API context. The argument to GrB_init defines the mode for the context. The two available modes are:

- GrB_BLOCKING: In this mode, each method in a sequence returns after its computations have completed and output arguments are available to subsequent statements in an application. When executing in GrB_BLOCKING mode, the methods execute in program order.
- GrB_NONBLOCKING: In this mode, methods in a sequence may return after arguments in the method have been tested for dimension and domain compatibility within the method but potentially before their computations complete. Output arguments are available to subsequent GraphBLAS methods in an application. When executing in GrB_NONBLOCKING mode, the methods in a sequence may execute in any order that preserves the mathematical result defined by the sequence.

An application can only create one context per execution instance.

4.1.2 finalize: Finalize a GraphBLAS context

Terminates and frees any internal resources created to support the GraphBLAS C API context.

C Syntax

```
GrB_Info GrB_finalize();
```

Return Values

GrB_SUCCESS	operation completed successfully.
GrB_PANIC	unknown internal error.

Description

Terminates and frees any internal resources created to support the GraphBLAS C API context. An application may not create a new context or call any other GraphBLAS methods after GrB_finalize has been called.

4.2 Object Methods

This section describes methods that setup and operate on GraphBLAS opaque objects but are not part of the the GraphBLAS math specification.

4.2.1 Algebra Methods

4.2.1.1 Type_new: Create a new GraphBLAS (user-defined) type

Creates a new user-defined GraphBLAS type. This type can then be used to create new operators, monoids, semirings, vectors and matrices.

C Syntax

```
GrB_Info GrB_Type_new(GrB_Type  *utype,
                      size_t     sizeof(ctype));
```

Parameters

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

ctype (IN) A C type that defines the new GraphBLAS user-defined type.

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 utype pointer is NULL.

Description

Given a C type ctype, this method returns in utype a handle to a new GraphBLAS type equivalent to that C type. Variables of this ctype must be a struct, union, or fixed-size array. In particular, given two variables, `src` and `dst`, of type ctype, the following operation must be a valid way to copy the contents of `src` to `dst`:

```
memcpy(&dst, &src, sizeof(ctype))
```

A new user-defined type utype should be destroyed with a call to GrB_free(utype) when no longer needed.

It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

4.2.1.2 UnaryOp_new: Create a new GraphBLAS unary operator

Initializes a new GraphBLAS unary operator with a specified user-defined function and its types (domains).

C Syntax

```
        GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,
                                 void       (*unary_func)(void*, const void*),
                                 GrB_Type     d_out,
                                 GrB_Type     d_in);
```

Parameters

unary_op (INOUT) On successful return, contains a handle to the newly created GraphBLAS unary operator object.

unary_func (IN) a pointer to a user-defined function that takes one input parameter of d_in's type and returns a value of d_out's type, both passed as void pointers. Specifically the signature of the function is expected to be of the form:

```
void func(void *out, const void *in);
```

d_out (IN) The GrB_Type of the return value of the unary operator being created. Should be one of the predefined GraphBLAS types in Table 2.2, or a user-defined GraphBLAS type.

d_in (IN) The GrB_Type of the input argument of the unary operator being created. Should be one of the predefined GraphBLAS types in Table 2.2, or a user-defined GraphBLAS type.

Return Values

GrB_SUCCESS operation completed successfully.

GrB_PANIC unknown internal error.

GrB_OUT_OF_MEMORY not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT any GrB_Type parameter (for user-defined types) has not been initialized by a call to GrB_Type_new.

GrB_NULL_POINTER unary_op or unary_func pointers are NULL.

Description

Creates a new GraphBLAS unary operator $f_u = \langle \mathbf{D}(\mathsf{d\_out}), \mathbf{D}(\mathsf{d\_in}), \mathsf{unary\_func}\rangle$ and returns a handle to it in unary_op.

It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

4.2.1.3 BinaryOp_new: Create a new GraphBLAS binary operator

Initializes a new GraphBLAS binary operator with a specified user-defined function and its types (domains).

C Syntax

```
GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,
                          void        (*binary_func)(void*,
                                                     const void*,
                                                     const void*),
                          GrB_Type      d_out,
                          GrB_Type      d_in1,
                          GrB_Type      d_in2);
```

Parameters

- **binary_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS binary operator object.
- **binary_func** (IN) A pointer to a user-defined function that takes two input parameters of types d_in1 and d_in2 and returns a value of type d_out, all passed as void pointers. Specifically the signature of the function is expected to be of the form:

```
void func(void *out, const void *in1, const void *in2);
```

- **d_out** (IN) The GrB_Type of the return value of the binary operator being created. Should be one of the predefined GraphBLAS types in Table 2.2, or a user-defined GraphBLAS type.
- **d_in1** (IN) The GrB_Type of the left hand argument of the binary operator being created. Should be one of the predefined GraphBLAS types in Table 2.2, or a user-defined GraphBLAS type.
- **d_in2** (IN) The GrB_Type of the right hand argument of the binary operator being created. Should be one of the predefined GraphBLAS types in Table 2.2, or a user-defined GraphBLAS type.

Return Values

GrB_SUCCESS	operation completed successfully.
GrB_PANIC	unknown internal error.
GrB_OUT_OF_MEMORY	not enough memory available for operation.
GrB_UNINITIALIZED_OBJECT	the GrB_Type (for user-defined types) has not been initialized by a call to GrB_Type_new.
GrB_NULL_POINTER	binary_op or binary_func pointer is NULL.

Description

Creates a new GraphBLAS binary operator $f_b = \langle \mathbf{D}(\mathsf{d\_out}), \mathbf{D}(\mathsf{d\_in1}), \mathbf{D}(\mathsf{d\_in2}), \mathsf{binary\_func} \rangle$ and returns a handle to it in binary_op.

It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

4.2.1.4 Monoid_new: Create new GraphBLAS monoid

Creates a new monoid with specified binary operator and identity value.

C Syntax

```
GrB_Info GrB_Monoid_new(GrB_Monoid    *monoid,
                        GrB_BinaryOp   binary_op,
                        <type>         identity);
```

Parameters

monoid	(INOUT) On successful return, contains a handle to the newly created GraphBLAS monoid object.
binary_op	(IN) An existing GraphBLAS associative binary operator whose input and output types are the same.
identity	(IN) The value of the identity element of the monoid. Must be the same type as the type used by the binary_op operator.

Return Values

GrB_SUCCESS	operation completed successfully.

GrB_PANIC	unknown internal error.
GrB_OUT_OF_MEMORY	not enough memory available for operation.
GrB_UNINITIALIZED_OBJECT	the GrB_BinaryOp has not been initialized by a call to GrB_BinaryOp_new.
GrB_NULL_POINTER	monoid pointer is NULL.
GrB_DOMAIN_MISMATCH	all three argument types of the binary operator and the type of the identity value are not the same.

Description

Creates a new monoid $M = \langle \mathbf{D}(\mathsf{binary\_op}), \mathsf{binary\_op}, \mathsf{identity} \rangle$ and returns a handle to it in monoid.

If binary_op is not associative, the results of GraphBLAS operations that require associativity of this monoid will be undefined.

It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

4.2.1.5 Semiring_new: Create new GraphBLAS semiring

Creates a new semiring with specified domain, operators, and elements.

C Syntax

```
GrB_Info GrB_Semiring_new(GrB_Semiring  *semiring,
                          GrB_Monoid     add_op,
                          GrB_BinaryOp   mul_op);
```

Parameters

semiring	(INOUT) On successful return, contains a handle to the newly created GraphBLAS semiring.
add_op	(IN) An existing GraphBLAS commutative monoid that specifies the addition operator and its identity.
mul_op	(IN) An existing GraphBLAS binary operator that specifies the semiring's multiplication operator. In addition, mul_op's output domain, $\mathbf{D}_{out}(\mathsf{mul\_op})$, must be the same as the add_op's domain $\mathbf{D}(\mathsf{add\_op})$.

Return Values

- GrB_SUCCESS operation completed successfully.
- GrB_PANIC unknown internal error.
- GrB_OUT_OF_MEMORY not enough memory available for this method to complete.
- GrB_UNINITIALIZED_OBJECT the add_op object has not been initialized with a call to GrB_Monoid_new or the mul_op object has not been not been initialized by a call to GrB_BinaryOp_new.
- GrB_NULL_POINTER semiring pointer is NULL.
- GrB_DOMAIN_MISMATCH the output domain of mul_op does not match the domain of the add_op monoid.

Description

Creates a new semiring $S = \langle \mathbf{D}_{out}(\mathsf{mul\_op}), \mathbf{D}_{in_1}(\mathsf{mul\_op}), \mathbf{D}_{in_2}(\mathsf{mul\_op}), \mathsf{add\_op}, \mathsf{mul\_op}, \mathbf{0}(\mathsf{add\_op})\rangle$ and returns a handle to it in semiring. Note that $\mathbf{D}_{out}(\mathsf{mul\_op})$ must be the same as $\mathbf{D}(\mathsf{add\_op})$.

If add_op is not commutative, then GraphBLAS operations using this semiring will be undefined.

It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

4.2.2 Vector Methods

4.2.2.1 Vector_new: Create new vector

Creates a new vector with specified domain and size.

C Syntax

```
        GrB_Info GrB_Vector_new(GrB_Vector *v,
                                GrB_Type    d,
                                GrB_Index   nsize);
```

Parameters

- v (INOUT) On successful return, contains a handle to the newly created GraphBLAS vector.
- d (IN) The type corresponding to the domain of the vector being created. Can be one of the predefined GraphBLAS types in Table 2.2, or an existing user-defined GraphBLAS type.

nsize (IN) The size of the vector being created.

Return Values

- GrB_SUCCESS: In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the API checks for the input arguments passed successfully. Either way, output vector v is ready to be used in the next method of the sequence.
- GrB_PANIC: Unknown internal error.
- GrB_INVALID_OBJECT: This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
- GrB_OUT_OF_MEMORY: Not enough memory available for operation.
- GrB_UNINITIALIZED_OBJECT: The GrB_Type object has not been initialized by a call to GrB_Type_new (needed for user-defined types).
- GrB_NULL_POINTER: The v pointer is NULL.
- GrB_INVALID_VALUE: nsize is zero.

Description

Creates a new vector $\mathbf{v}$ of domain $\mathbf{D}(\mathsf{d})$, size nsize, and empty $\mathbf{L}(\mathbf{v})$. The method returns a handle to the new vector in v.

It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

4.2.2.2 Vector_dup: Create a copy of a GraphBLAS vector

Creates a new vector with the same domain, size, and contents as another vector.

C Syntax

```
GrB_Info GrB_Vector_dup(GrB_Vector       *w,
                        const GrB_Vector  u);
```

Parameters

w (INOUT) On successful return, contains a handle to the newly created GraphBLAS vector.

u (IN) The GraphBLAS vector to be duplicated.

Return Values

GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the API checks for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
GrB_OUT_OF_MEMORY	Not enough memory available for operation.
GrB_UNINITIALIZED_OBJECT	The GraphBLAS vector, u, has not been initialized by a call to Vector_new or Vector_dup.
GrB_NULL_POINTER	The w pointer is NULL.

Description

Creates a new vector **w** of domain $\mathbf{D}(\mathsf{u})$, size $\mathbf{size}(\mathsf{u})$, and contents $\mathbf{L}(\mathsf{u})$. The method returns a handle to the new vector in w.

It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

4.2.2.3 Vector_clear: Clear a vector

Removes all the elements (tuples) from a vector.

C Syntax

```
GrB_Info GrB_Vector_clear(GrB_Vector v);
```

Parameters

v (INOUT) An existing GraphBLAS vector to clear.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the API checks for the input arguments passed successfully. Either way, output vector v is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to Vector_new or Vector_dup.

Description

Removes all elements (tuples) from an existing vector. After the call to GrB_Vector_clear(v), $\mathbf{L}(\mathbf{v}) = \emptyset$. The size of the vector does not change.

4.2.2.4 Vector_size: Size of a vector

Retrieve the size of a vector.

C Syntax

```
GrB_Info GrB_Vector_size(GrB_Index        *nsize,
                         const GrB_Vector  v);
```

Parameters

nsize (OUT) On successful return, is set to the size of the vector.

v (IN) An existing GraphBLAS vector being queried.

Return Values

GrB_SUCCESS In blocking or non-blocking mode, the operation completed successfully and the value of nsize has been set.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to Vector_new or Vector_dup.

GrB_NULL_POINTER nsize pointer is NULL.

Description

Return **size**(v) in nsize.

4.2.2.5 Vector_nvals: Number of stored elements in a vector

Retrieve the number of stored elements (tuples) in a vector.

C Syntax

```
GrB_Info GrB_Vector_nvals(GrB_Index        *nvals,
                          const GrB_Vector  v);
```

Parameters

nvals (OUT) On successful return, this is set to the number of stored elements (tuples) in the vector.

v (IN) An existing GraphBLAS vector being queried.

Return Values

GrB_SUCCESS In blocking or non-blocking mode, the operation completed successfully and the value of nvals has been set.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to Vector_new or Vector_dup.

GrB_NULL_POINTER The nvals pointer is NULL.

Description

Return **nvals**(v) in nvals. This is the number of stored elements in vector v, which is the size of $\mathbf{L}(\mathbf{v})$ (see Section 3.4).

4.2.2.6 Vector_build: Store elements from tuples into a vector

C Syntax

```
GrB_Info GrB_Vector_build(GrB_Vector             w,
                          const GrB_Index       *indices,
                          const <type>          *values,
                          GrB_Index              n,
                          const GrB_BinaryOp     dup);
```

Parameters

w (INOUT) An existing Vector object to store the result.

indices (IN) Pointer to an array of indices.

values (IN) Pointer to an array of scalars of a type that is compatible with the domain of vector w.

n (IN) The number of entries contained in each array (the same for indices and values).

dup (IN) An associative and commutative binary operator to apply when duplicate values for the same location are present in the input arrays. All three domains of dup must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the API checks for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT Either w has not been initialized by a call to by GrB_Vector_new or by GrB_Vector_dup, or dup has not been initialized by a call to by GrB_BinaryOp_new.

GrB_NULL_POINTER indices or values pointer is NULL.

GrB_INDEX_OUT_OF_BOUNDS A value in indices is outside the allowed range for w.

GrB_DOMAIN_MISMATCH Either the domains of the GraphBLAS binary operator dup are not all the same, or the domains of values and w are incompatible with each other or D_{dup}.

GrB_OUTPUT_NOT_EMPTY Output vector w already contains valid tuples (elements). In other words, GrB_Vector_nvals(C) returns a positive value.

Description

An internal vector $\widetilde{\mathbf{w}} = \langle D_{dup}, \mathbf{size}(\mathsf{w}), \emptyset \rangle$ is created, which only differs from w in its domain.

Each tuple $\{\mathsf{indices[k]}, \mathsf{values[k]}\}$, where $0 \leq k < \mathsf{n}$, is a contribution to the output in the form of

$$\widetilde{\mathbf{w}}(\mathsf{indices[k]}) = (D_{dup})\,\mathsf{values[k]}.$$

If multiple values for the same location are present in the input arrays, the dup binary operand is used to reduce them before assignment into $\widetilde{\mathbf{w}}$ as follows:

$$\widetilde{\mathbf{w}}_i = \bigoplus_{k:\,\mathsf{indices[k]}=i} (D_{dup})\,\mathsf{values[k]},$$

where $\oplus$ is the dup binary operator. Finally, the resulting $\widetilde{\mathbf{w}}$ is copied into w via typecasting its values to $\mathbf{D}(\mathsf{w})$ if necessary. If $\oplus$ is not associative or not commutative, the result is undefined.

The nonopaque input arrays, indices and values, must be at least as large as n.

It is an error to call this function on an output object with existing elements. In other words, GrB_Vector_nvals(w) should evaluate to zero prior to calling this function.

After GrB_Vector_build returns, it is safe for a programmer to modify or delete the arrays indices or values.

4.2.2.7 Vector_setElement: Set a single element in a vector

Set one element of a vector to a given value.

C Syntax

```
GrB_Info GrB_Vector_setElement(GrB_Vector   w,
                               <type>       val,
                               GrB_Index    index);
```

Parameters

- w (INOUT) An existing GraphBLAS vector for which an element is to be assigned.
- val (IN) Scalar value to assign. The type must be compatible with the domain of w.
- index (IN) The location of the element to be assigned.

Return Values

- GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on index/dimensions and domains for the input arguments passed successfully. Either way, the output vector w is ready to be used in the next method of the sequence.
- GrB_PANIC Unknown internal error.
- GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
- GrB_OUT_OF_MEMORY Not enough memory available for operation.
- GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, w, has not been initialized by a call to Vector_new or Vector_dup.
- GrB_INVALID_INDEX index specifies a location that is outside the dimensions of w.
- GrB_DOMAIN_MISMATCH The domains of w and val are incompatible.

Description

First, the scalar and output vector are tested for domain compatibility as follows: $\mathbf{D}(\mathsf{val})$ must be compatible with $\mathbf{D}(\mathsf{w})$. Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_Vector_setElement ends and the domain mismatch error listed above is returned.

Then, the index parameter is checked for a valid value where the following condition must hold:

$$0 \leq \mathsf{index} < \mathbf{size}(\mathsf{w})$$

If this condition is violated, execution of GrB_Vector_extractElement ends and the invalid index error listed above is returned.

We are now ready to carry out the assignment val; that is:

$$\mathsf{w}(\mathsf{index}) = \mathsf{val}$$

If a value existed at this location in w, it will be overwritten; otherwise, and new value is stored in w.

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents of w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

4.2.2.8 Vector_extractElement: Extract a single element from a vector.

Extract one element of a vector into a scalar.

C Syntax

```
GrB_Info GrB_Vector_extractElement(<type>           *val,
                                   const GrB_Vector  u,
                                   GrB_Index         index);
```

Parameters

- **val** (INOUT) Pointer to a scalar of type that is compatible with the domain of vector w. On successful return, this scalar holds the result of the operation. Any previous value in val is overwritten.
- **u** (IN) The GraphBLAS vector from which an element is extracted.
- **index** (IN) The location in u to extract.

Return Values

GrB_SUCCESS In blocking or non-blocking mode, the operation completed successfully. This indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully, and the output scalar, val, has been computed and is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, u, has not been initialized by a call to Vector_new or Vector_dup.

GrB_NULL_POINTER val pointer is NULL.

GrB_NO_VALUE There is no stored value at specified location.

GrB_INVALID_INDEX index specifies a location that is outside the dimensions of w.

GrB_DOMAIN_MISMATCH The domains of the vector or scalar are incompatible.

Description

First, the scalar and input vector are tested for domain compatibility as follows: $\mathbf{D}(\mathsf{val})$ must be compatible with $\mathbf{D}(\mathsf{u})$. Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_Vector_extractElement ends and the domain mismatch error listed above is returned.

Then, the index parameter is checked for a valid value where the following condition must hold:

$$0 \leq \mathsf{index} < \mathbf{size}(\mathsf{u})$$

If this condition is violated, execution of GrB_Vector_extractElement ends and the invalid index error listed above is returned.

We are now ready to carry out the extract into the output argument, val; that is:

$$\mathsf{val} = \mathsf{u}(\mathsf{index})$$

where the following condition must be true:

$$\mathsf{index} \in \mathbf{ind}(\mathsf{u})$$

If this condition is violated, execution of GrB_Vector_extractElement ends and the "no value" error listed above is returned.

In both GrB_BLOCKING mode GrB_NONBLOCKING mode if the method exits with return value GrB_SUCCESS, the new contents of val are as defined above. In other words, the method does not return until any operations required to fully compute the GraphBLAS vector u have completed.

In GrB_NONBLOCKING mode, if the return value is not GrB_SUCCESS, an error in a method occurring earlier in the sequence may have occurred that prevents completion of the GraphBLAS vector u. The GrB_error() method should be called for additional information about these errors.

4.2.2.9 Vector_extractTuples: Extract tuples from a vector

Extract the contents of a GraphBLAS vector into non-opaque data structures.

C Syntax

```
GrB_Info GrB_Vector_extractTuples(GrB_Index            *indices,
                                   <type>               *values,
                                   GrB_Index            *n,
                                   const GrB_Vector      v);
```

- **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored values' indices.
- **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of the stored values whose type is compatible with $\mathbf{D}(\mathbf{v})$.
- **n** (INOUT) Pointer to a value indicating (on input) the number of elements the values and indices arrays can hold. Upon return, it will contain the number of values written to the arrays.
- **v** (IN) An existing GraphBLAS vector.

Return Values

- **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed successfully. This indicates that the compatibility tests on the input argument passed successfully, and the output arrays, indices and values, have been computed.
- **GrB_PANIC** Unknown internal error.
- **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused

by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_INSUFFICIENT_SPACE Not enough space in indices and values (as indicated by the n parameter) to hold all of the tuples that will be extacted.

GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to Vector_new or Vector_dup.

GrB_NULL_POINTER indices, values, or n pointer is NULL.

GrB_DOMAIN_MISMATCH The domains of the v vector or values array are incompatible with one another.

Description

This method will extract all the tuples from the GraphBLAS vector v. The values associated with those tuples are placed in the values array and the indices are placed in the indices array. Both indices and values must be pre-allocated by the user to have enough space to hold at least GrB_Vector_nvals(v) elements before calling this function.

Upon return of this function, n will be set to the number of values (and indices) copied. Also, the entries of indices are unique, but not necessarily sorted. Each tuple (i, v_i) in v is unzipped and copied into a distinct kth location in output vectors:

$$\{\mathsf{indices[k]}, \mathsf{values[k]}\} \leftarrow (i, v_i),$$

where $0 \leq k <$ GrB_Vector_nvals(v). No gaps in output vectors are allowed; that is, if indices[k] and values[k] exist upon return, so does indices[j] and values[j] for all j such that $0 \leq j < k$.

Note that if the value in n on input is less than the number of values contained in the vector v, then a GrB_INSUFFICIENT_SPACE error is returned because it is undefined which subset of values would be extracted otherwise.

In both GrB_BLOCKING mode GrB_NONBLOCKING mode if the method exits with return value GrB_SUCCESS, the new contents of the arrays indices and values are as defined above. In other words, the method does not return until any operations required to fully compute the GraphBLAS vector v have completed.

In GrB_NONBLOCKING mode, if the return value is not GrB_SUCCESS, an error in a method occurring earlier in the sequence may have occurred that prevents completion of the GraphBLAS vector v. The GrB_error() method should be called for additional information about these errors.

4.2.3 Matrix Methods

4.2.3.1 Matrix_new: Create new matrix

Creates a new matrix with specified domain and dimensions.

C Syntax

```
GrB_Info GrB_Matrix_new(GrB_Matrix *A,
                        GrB_Type    d,
                        GrB_Index   nrows,
                        GrB_Index   ncols);
```

Parameters

- A (INOUT) On successful return, contains a handle to the newly created GraphBLAS matrix.
- d (IN) The type corresponding to the domain of the matrix being created. Can be one of the predefined GraphBLAS types in Table 2.2, or an existing user-defined GraphBLAS type.
- nrows (IN) The number of rows of the matrix being created.
- ncols (IN) The number of columns of the matrix being created.

Return Values

- GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the API checks for the input arguments passed successfully. Either way, output matrix A is ready to be used in the next method of the sequence.
- GrB_PANIC Unknown internal error.
- GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
- GrB_OUT_OF_MEMORY Not enough memory available for operation.
- GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to GrB_Type_new (needed for user-defined types).
- GrB_NULL_POINTER The A pointer is NULL.
- GrB_INVALID_VALUE nrows or ncols is zero.

Description

Creates a new matrix $\mathbf{A}$ of domain $\mathbf{D}(\mathsf{d})$, size nrows $\times$ ncols, and empty $\mathbf{L}(\mathbf{A})$. The method returns a handle to the new matrix in A.

It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

4.2.3.2 Matrix_dup: Create a copy of a GraphBLAS matrix

Creates a new matrix with the same domain, dimensions, and contents as another matrix.

C Syntax

```
GrB_Info GrB_Matrix_dup(GrB_Matrix       *C,
                        const GrB_Matrix  A);
```

Parameters

- C (INOUT) On successful return, contains a handle to the newly created GraphBLAS matrix.
- A (IN) The GraphBLAS matrix to be duplicated.

Return Values

GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the API checks for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
GrB_OUT_OF_MEMORY	Not enough memory available for operation.
GrB_UNINITIALIZED_OBJECT	The GraphBLAS matrix, A, has not been initialized by a call to Matrix_new or Matrix_dup.
GrB_NULL_POINTER	The C pointer is NULL.

Description

Creates a new matrix **C** of domain $\mathbf{D}(\mathsf{A})$, size $\mathbf{nrows}(\mathsf{A}) \times \mathbf{ncols}(\mathsf{A})$, and contents $\mathbf{L}(\mathsf{A})$. It returns a handle to it in C.

It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

4.2.3.3 Matrix_clear: Clear a matrix

Removes all elements (tuples) from a matrix.

C Syntax

```
GrB_Info GrB_Matrix_clear(GrB_Matrix A);
```

Parameters

A (IN) An exising GraphBLAS matrix to clear.

Return Values

GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the API checks for the input arguments passed successfully. Either way, output matrix A is ready to be used in the next method of the sequence.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
GrB_OUT_OF_MEMORY	Not enough memory available for operation.
GrB_UNINITIALIZED_OBJECT	The GraphBLAS matrix, *A, has not been initialized by a call to Matrix_new or Matrix_dup.

Description

Removes all elements (tuples) from an existing matrix. After the call to GrB_Matrix_clear(A), $\mathbf{L}(\mathbf{A}) = \emptyset$. The dimensions of the matrix do not change.

4.2.3.4 Matrix_nrows: Number of rows in a matrix

Retrieve the number of rows in a matrix.

C Syntax

```
        GrB_Info GrB_Matrix_nrows(GrB_Index        *nrows,
                                  const GrB_Matrix  A);
```

Parameters

nrows (OUT) On successful return, contains the number of rows in the matrix.

A (IN) An existing GraphBLAS matrix being queried.

Return Values

GrB_SUCCESS In blocking or non-blocking mode, the operation completed successfully and the value of nrows has been set.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to Matrix_new or Matrix_dup.

GrB_NULL_POINTER nrows pointer is NULL.

Description

Return **nrows**(A) in nrows (the number of rows).

4.2.3.5 Matrix_ncols: Number of columns in a matrix

Retrieve the number of columns in a matrix.

C Syntax

```
        GrB_Info GrB_Matrix_ncols(GrB_Index        *ncols,
                                  const GrB_Matrix  A);
```

Parameters

ncols (OUT) On successful return, contains the number of columns in the matrix.

A (IN) An existing GraphBLAS matrix being queried.

Return Values

GrB_SUCCESS In blocking or non-blocking mode, the operation completed successfully and the value of ncols has been set.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to Matrix_new or Matrix_dup.

GrB_NULL_POINTER ncols pointer is NULL.

Description

Return **ncols**(A) in ncols (the number of columns).

4.2.3.6 Matrix_nvals: Number of stored elements in a matrix

Retrieve the number of stored elements (tuples) in a matrix.

C Syntax

```
GrB_Info GrB_Matrix_nvals(GrB_Index        *nvals,
                          const GrB_Matrix  A);
```

Parameters

nvals (OUT) On successful return, contains the number of stored elements (tuples) in the matrix.

A (IN) An existing GraphBLAS matrix being queried.

Return Values

GrB_SUCCESS In blocking or non-blocking mode, the operation completed successfully and the value of nvals has been set.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to Matrix_new or Matrix_dup.

GrB_NULL_POINTER The nvals pointer is NULL.

Description

Return **nvals**(A) in nvals. This is the number of tuples stored in matrix A, which is the size of $\mathbf{L}(\mathbf{A})$ (see Section 3.5).

4.2.3.7 Matrix_build: Store elements from tuples into a matrix

C Syntax

```
GrB_Info GrB_Matrix_build(GrB_Matrix              C,
                          const GrB_Index        *row_indices,
                          const GrB_Index        *col_indices,
                          const <type>           *values,
                          GrB_Index               n,
                          const GrB_BinaryOp      dup);
```

Parameters

C (INOUT) An existing Matrix object to store the result.

row_indices (IN) Pointer to an array of row indices.

col_indices (IN) Pointer to an array of column indices.

values (IN) Pointer to an array of scalars of a type that is compatible with the domain of matrix, C.

n (IN) The number of entries contained in each array (the same for row_indices, col_indices, and values).

dup (IN) An associative and commutative binary function to apply when duplicate values for the same location are present in the input arrays. All three domains of dup must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the API checks for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT Either C has not been initialized by a call to by GrB_Matrix_new or by GrB_Matrix_dup, or dup has not been initialized by a call to by GrB_BinaryOp_new.

GrB_NULL_POINTER row_indices, col_indices or values pointer is NULL.

GrB_INDEX_OUT_OF_BOUNDS A value in row_indices or col_indices is outside the allowed range for C.

GrB_DOMAIN_MISMATCH Either the domains of the GraphBLAS binary operator dup are not all the same, or the domains of values and C are incompatible with each other or D_{dup}.

GrB_OUTPUT_NOT_EMPTY Output matrix C already contains valid tuples (elements). In other words, GrB_Matrix_nvals(C) returns a positive value.

Description

An internal matrix $\widetilde{\mathbf{C}} = \langle D_{dup}, \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \emptyset \rangle$ is created, which only differs from C in its domain.

Each tuple $\{\mathsf{row\_indices[k]}, \mathsf{col\_indices[k]}, \mathsf{values[k]}\}$, where $0 \leq k < \mathsf{n}$, is a contribution to the output in the form of

$$\widetilde{\mathbf{C}}(\mathsf{row\_indices[k]}, \mathsf{col\_indices[k]}) = (D_{dup})\, \mathsf{values[k]}.$$

If multiple values for the same location are present in the input arrays, the dup binary operand is used to reduce them before assignment into $\widetilde{\mathbf{C}}$ as follows:

$$\widetilde{\mathbf{C}}_{ij} = \bigoplus_{k:\ \mathsf{row\_indices[k]}=i\ \wedge\ \mathsf{col\_indices[k]}=j} (D_{dup})\,\mathsf{values[k]},$$

where $\oplus$ is the dup binary operator. Finally, the resulting $\widetilde{\mathbf{C}}$ is copied into C via typecasting its values to $\mathbf{D}$(C) if necessary. If $\oplus$ is not associative or not commutative, the result is undefined.

The nonopaque input arrays row_indices, col_indices, and values must be at least as large as n.

It is an error to call this function on an output object with existing elements. In other words, GrB_Matrix_nvals(C) should evaluate to zero prior to calling this function.

After GrB_Matrix_build returns, it is safe for a programmer to modify or delete the arrays row_indices, col_indices, or values.

4.2.3.8 Matrix_setElement: Set a single element in matrix

Set one element of a matrix to a given value.

C Syntax

```
GrB_Info GrB_Matrix_setElement(GrB_Matrix   C,
                               <type>       val,
                               GrB_Index    row_index,
                               GrB_Index    col_index);
```

Parameters

- **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.
- **val** (IN) Scalar value to assign. The type must be compatible with the domain of C.
- **row_index** (IN) Row index of element to be assigned
- **col_index** (IN) Column index of element to be assigned

Return Values

- **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on index/dimensions and domains for the input arguments passed successfully. Either way, the output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to Matrix_new or Matrix_dup.

GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e., not less than **nrows**(C) or **ncols**(C), respectively).

GrB_DOMAIN_MISMATCH The domains of C and val are incompatible.

Description

First, the scalar and output matrix are tested for domain compatibility as follows: $\mathbf{D}(\mathsf{val})$ must be compatible with $\mathbf{D}(\mathsf{C})$. Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_Matrix_extractElement ends and the domain mismatch error listed above is returned.

Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 0 \ \leq \ \mathsf{row\_index} \ &< \ \mathbf{nrows}(\mathsf{C}), \\ 0 \ \leq \ \mathsf{col\_index} \ &< \ \mathbf{ncols}(\mathsf{C}) \end{aligned}$$

If either of these conditions is violated, execution of GrB_Matrix_extractElement ends and the invalid index error listed above is returned.

We are now ready to carry out the assignment of val; that is,

$$\mathsf{C}(\mathsf{row\_index}, \mathsf{col\_index}) = \mathsf{val}$$

If a value existed at this location in C, it will be overwritten; otherwise, and new value is stored in C.

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents of C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector C is as defined above but may not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

4.2.3.9 Matrix_extractElement: Extract a single element from a matrix

Extract one element of a matrix into a scalar.

C Syntax

```
GrB_Info GrB_Matrix_extractElement(<type>           *val,
                                   const GrB_Matrix  A,
                                   GrB_Index         row_index,
                                   GrB_Index         col_index);
```

Parameters

- **val** (OUT) Pointer to a scalar of type that is compatible with the domain of matrix A. On successful return, this scalar holds the result of the operation. Any previous value in val is overwritten.
- **A** (IN) The GraphBLAS matrix from which an element is extracted.
- **row_index** (IN) The row index of location in A to extract.
- **col_index** (IN) The column index of location in A to extract.

Return Values

- **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed successfully. This indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully, and the output scalar, val, has been computed and is ready to be used in the next method of the sequence.
- **GrB_PANIC** Unknown internal error.
- **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
- **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
- **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, A, has not been initialized by a call to Matrix_new or Matrix_dup.
- **GrB_NULL_POINTER** val pointer is NULL.
- **GrB_NO_VALUE** There is no stored value at specified location.
- **GrB_INVALID_INDEX** row_index or col_index is outside the allowable range (i.e. less than zero or greater than or equal to **nrows**(A) or **ncols**(A), respectively).
- **GrB_DOMAIN_MISMATCH** The domains of the matrix and scalar are incompatible.

Description

First, the scalar and input matrix are tested for domain compatibility as follows: $\mathbf{D}(\mathsf{val})$ must be compatible with $\mathbf{D}(\mathsf{A})$. Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_Matrix_extractElement ends and the domain mismatch error listed above is returned.

Then, both index parameters are checked for valid values where following conditions must hold:

$$0 \le \mathsf{row\_index} < \mathbf{nrows}(\mathsf{A}),$$
$$0 \le \mathsf{col\_index} < \mathbf{ncols}(\mathsf{A})$$

If either of these conditions is violated, execution of GrB_Matrix_extractElement ends and the invalid index error listed above is returned.

We are now ready to carry out the extract into the output argument, val; that is,

$$\mathsf{val} = \mathsf{A}(\mathsf{row\_index}, \mathsf{col\_index})$$

where the following condition must be true:

$$(\mathsf{row\_index}, \mathsf{col\_index}) \in \mathbf{ind}(\mathsf{A})$$

If this condition is violated, execution of GrB_Matrix_extractElement ends and the "no value" error listed above is returned.

In both GrB_BLOCKING mode GrB_NONBLOCKING mode if the method exits with return value GrB_SUCCESS, the new contents of val are as defined above. In other words, the method does not return until any operations required to fully compute the GraphBLAS matrix A have completed.

In GrB_NONBLOCKING mode, if the return value is other than GrB_SUCCESS, an error in a method occurring earlier in the sequence may have occurred that prevents completion of the GraphBLAS matrix A. The GrB_error() method should be called for additional information about such errors.

4.2.3.10 Matrix_extractTuples: Extract tuples from a matrix

Extract the contents of a GraphBLAS matrix into non-opaque data structures.

C Syntax

```
GrB_Info GrB_Matrix_extractTuples(GrB_Index            *row_indices,
                                  GrB_Index            *col_indices,
                                  <type>               *values,
                                  GrB_Index            *n,
                                  const GrB_Matrix      A);
```

Parameters

row_indices (OUT) Pointer to an array of row indices that is large enough to hold all of the row indices.

col_indices (OUT) Pointer to an array of column indices that is large enough to hold all of the column indices.

values (OUT) Pointer to an array of scalars of a type that is large enough to hold all of the stored values whose type is compatible with $\mathbf{D}(\mathbf{A})$.

n (INOUT) Pointer to a value indicating (in input) the number of elements the values, row_indices, and col_indices arrays can hold. Upon return, it will contain the number of values written to the arrays.

A (IN) An existing GraphBLAS matrix.

Return Values

GrB_SUCCESS In blocking or non-blocking mode, the operation completed successfully. This indicates that the compatibility tests on the input argument passed successfully, and the output arrays, indices and values, have been computed.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_INSUFFICIENT_SPACE Not enough space in row_indices, col_indices, and values (as indicated by the n parameter) to hold all of the tuples that will be extacted.

GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to Matrix_new or Matrix_dup.

GrB_NULL_POINTER row_indices, col_indices, values or n pointer is NULL.

GrB_DOMAIN_MISMATCH The domains of the A matrix and values array are incompatible with one another.

Description

This method will extract all the tuples from the GraphBLAS matrix A. The values associated with those tuples are placed in the values array, the column indices are placed in the col_indices array, and the row indices are placed in the row_indices array. These output arrays are pre-allocated by the user before calling this function such that each output array has enough space to hold at least GrB_Matrix_nvals(A) elements.

Upon return of this function, a pair of $\{\mathsf{row\_indices[k]}, \mathsf{col\_indices[k]}\}$ are unique for every valid k, but they are not required to be sorted in any particular order. Each tuple (i, j, A_{ij}) in A is unzipped and copied into a distinct kth location in output vectors:

$$\{\mathsf{row\_indices[k]}, \mathsf{col\_indices[k]}, \mathsf{values[k]}\} \leftarrow (i, j, A_{ij}),$$

where $0 \leq k <$ GrB_Matrix_nvals(v). No gaps in output vectors are allowed; that is, if row_indices[k], col_indices[k] and values[k] exist upon return, so does row_indices[j], col_indices[j] and values[j] for all j such that $0 \leq j < k$.

Note that if the value in n on input is less than the number of values contained in the matrix A, then a GrB_INSUFFICIENT_SPACE error is returned since it is undefined which subset of values would be extracted.

In both GrB_BLOCKING mode GrB_NONBLOCKING mode if the method exits with return value GrB_SUCCESS, the new contents of the arrays row_indices, col_indices and values are as defined above. In other words, the method does not return until any operations required to fully compute the GraphBLAS vector A have completed.

In GrB_NONBLOCKING mode, if the return value is not GrB_SUCCESS, an error in a method occurring earlier in the sequence may have occurred that prevents completion of the GraphBLAS vector A. The GrB_error() method should be called for additional information about these errors.

4.2.4 Descriptor Methods

The methods in this section create and set values in descriptors. A descriptor is an opaque GraphBLAS object the values of which are used to modify the behavior of GraphBLAS operations.

4.2.4.1 Descriptor_new: Create new descriptor

Creates a new (empty or default) descriptor.

C Syntax

```
GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);
```

Parameters

desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS descriptor.

Return Value

GrB_SUCCESS	The method completed successfully.
GrB_PANIC	unknown internal error.
GrB_OUT_OF_MEMORY	not enough memory available for operation.
GrB_NULL_POINTER	desc pointer is NULL.

Description

Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can be populated by calls to Descriptor_set.

It is not an error to call this method more than once on the same variable; however, the handle to the previously created object will be overwritten.

4.2.4.2 Descriptor_set: Set content of descriptor

Sets the content for a field for an existing descriptor.

C Syntax

```
GrB_Info GrB_Descriptor_set(GrB_Descriptor      desc,
                            GrB_Desc_Field      field,
                            GrB_Desc_Value      val);
```

Parameters

desc	(IN) An existing GraphBLAS descriptor to be modified.
field	(IN) The field being set.
val	(IN) New value for the field being set.

Return Values

GrB_SUCCESS	operation completed successfully.
GrB_PANIC	unknown internal error.
GrB_OUT_OF_MEMORY	not enough memory available for operation.
GrB_UNINITIALIZED_OBJECT	the desc parameter has not been initialized by a call to new.
GrB_INVALID_VALUE	invalid value set on the field, or invalid field.

Description

For a given descriptor, the GrB_Descriptor_set method can be called for each field in the descriptor to set the value associated with that field. Valid values for the field parameter include the following:

- **GrB_OUTP** refers to the output parameter (result) of the operation.
- **GrB_MASK** refers to the mask parameter of the operation.
- **GrB_INP0** refers to the first input parameters of the operation (matrices and vectors).
- **GrB_INP1** refers to the second input parameters of the operation (matrices and vectors).

Valid values for the val parameter are:

- **GrB_SCMP** Use the structural complement of the corresponding mask (GrB_MASK) parameter.
- **GrB_TRAN** Use the transpose of the corresponding matrix parameter (valid for input matrix parameters only).
- **GrB_REPLACE** When assigning the masked values to the output matrix or vector, clear the matrix first (or clear the non-masked entries). The default behavior is to leave non-masked locations unchanged. Valid for the GrB_OUTP parameter only.

A value for a given field may be set multiple times. For a sequence of calls to the GrB_Descriptor_set method, the final call encountered in program order overwrites prior values to define the observed value for that field. Fields that are not set have their default value, as defined in Section 3.7.

4.2.5 free method

Destroys a previously created GraphBLAS object and releases any resources associated with the object.

C Syntax

```
GrB_Info GrB_free(GrB_Object *obj);
```

Parameters

- **obj** (INOUT) An existing GraphBLAS object to be destroyed. Can be any of the opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op, or type. On successful completion of GrB_free, obj behaves as an uninitialized object.

Return Values

GrB_SUCCESS	operation completed successfully
GrB_PANIC	unknown internal error. If this return value is encountered when in nonblocking mode, the error responsible for the panic condition could be from any method involved in the computation of the input object. The GrB_error() method should be called for additional information.

Description

GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime system. A call to GrB_free frees those resources so they are available for use by other GraphBLAS objects.

The parameter passed into GrB_free is a handle referencing a GraphBLAS opaque object of a data type from table 2.1. After the GrB_free method returns, the object referenced by the input handle is destroyed and the handle has the value GrB_INVALID_HANDLE. The handle can be used in subsequent GraphBLAS methods but only after the handle has been reinitialized with a call the the appropriate _new or _dup method.

Note that unlike other GraphBLAS methods, calling GrB_free with an object with an invalid handle is legal. The system may attempt to free resources that might be associated with that object, if possible, and return normally.

When using GrB_free it is possible to create a dangling reference to an object. This would occur when a handle is assigned to a second variable of the same opaque type. This creates two handles that reference the same object. If GrB_free is called with one of the variables, the object is destroyed and the handle associated with the other variable no longer references a valid object. This is not an error condition that the implementation of the GraphBLAS API can be expected to catch, hence programmers must take care to prevent this situation from occurring.

4.3 GraphBLAS Operations

The GraphBLAS operations are defined in the GraphBLAS math specification and summarized in Table 4.1. In addition to methods that implement these fundamental GraphBLAS operations, we support a number of variants that have been found to be especially useful in algorithm development. A flowchart of the overall behavior of a GraphBLAS operation is shown in Figure 4.1.

Domains and Casting

A GraphBLAS operation is only valid when the domains of the GraphBLAS objects are mathematically consistent. The C programming language defines implicit casts between built-in data types. For example, floats, doubles, and ints can be freely mixed according to the rules defined for implicit

Table 4.1: A mathematical notation for the fundamental GraphBLAS operations supported in this specification. Input matrices $\mathbf{A}$ and $\mathbf{B}$ may be optionally transposed (not shown). Use of an optional accumulate with existing values in the output object is indicated with $\odot$. Use of optional write masks and replace flags are indicated as $\mathbf{C}\langle\mathbf{M}, z\rangle$ when applied to the output matrix, $\mathbf{C}$. The mask or its structural complement (not shown) controls which values resulting from the operation on the right-hand side are written into the output object. The "replace" option, indicated by specifying the z flag, means that all values in the output object are removed prior to assignment. If "replace" is not specifed, only the values/locations computed on the right-hand side and allowed by the mask will be written to the output ("merge" mode).

Operation Name	Mathematical Notation
mxm	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot \mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot \mathbf{A} \oplus . \otimes \mathbf{u}$
vxm	$\mathbf{w}^T\langle\mathbf{m}^T, z\rangle = \mathbf{w}^T \odot \mathbf{u}^T \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot \mathbf{u} \otimes \mathbf{v}$
eWiseAdd	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot \mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot \mathbf{u} \oplus \mathbf{v}$
reduce (row)	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$
reduce (scalar)	$s = s \odot [\oplus_{i,j} \mathbf{A}(i, j)]$
	$s = s \odot [\oplus_i \mathbf{u}(i)]$
apply	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot f_u(\mathbf{A})$
	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot f_u(\mathbf{u})$
transpose	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot \mathbf{A}^T$
extract	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot \mathbf{A}(\boldsymbol{i}, \boldsymbol{j})$
	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot \mathbf{u}(\boldsymbol{i})$
assign	$\mathbf{C}\langle\mathbf{M}, z\rangle(\boldsymbol{i}, \boldsymbol{j}) = \mathbf{C}(\boldsymbol{i}, \boldsymbol{j}) \odot \mathbf{A}$
	$\mathbf{w}\langle\mathbf{m}, z\rangle(\boldsymbol{i}) = \mathbf{w}(\boldsymbol{i}) \odot \mathbf{u}$

casts. It is the responsibility of the user to assure that these casts are appropriate for the algorithm in question. For example, a cast to `int` implies truncation of a floating point type. Depending on the operation, this truncation error could lead to erroneous results. Furthermore, casting a wider type onto a narrower type can lead to overflow errors. The GraphBLAS operations do not attempt to protect a user from these sorts of errors.

When user-define types are involved, however, GraphBLAS requires strict equivalence between types and no casting is supported. If GraphBLAS detects these mismatches, it will return a domain mismatch error.

Dimensions and Transposes

GraphBLAS operations also make assumptions about the numbers of dimensions and sizes of vectors and matrices in an operation. An operation will test these sizes and report an error if they are not *shape compatible*. For example, when multiplying two matrices, $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, the number of rows

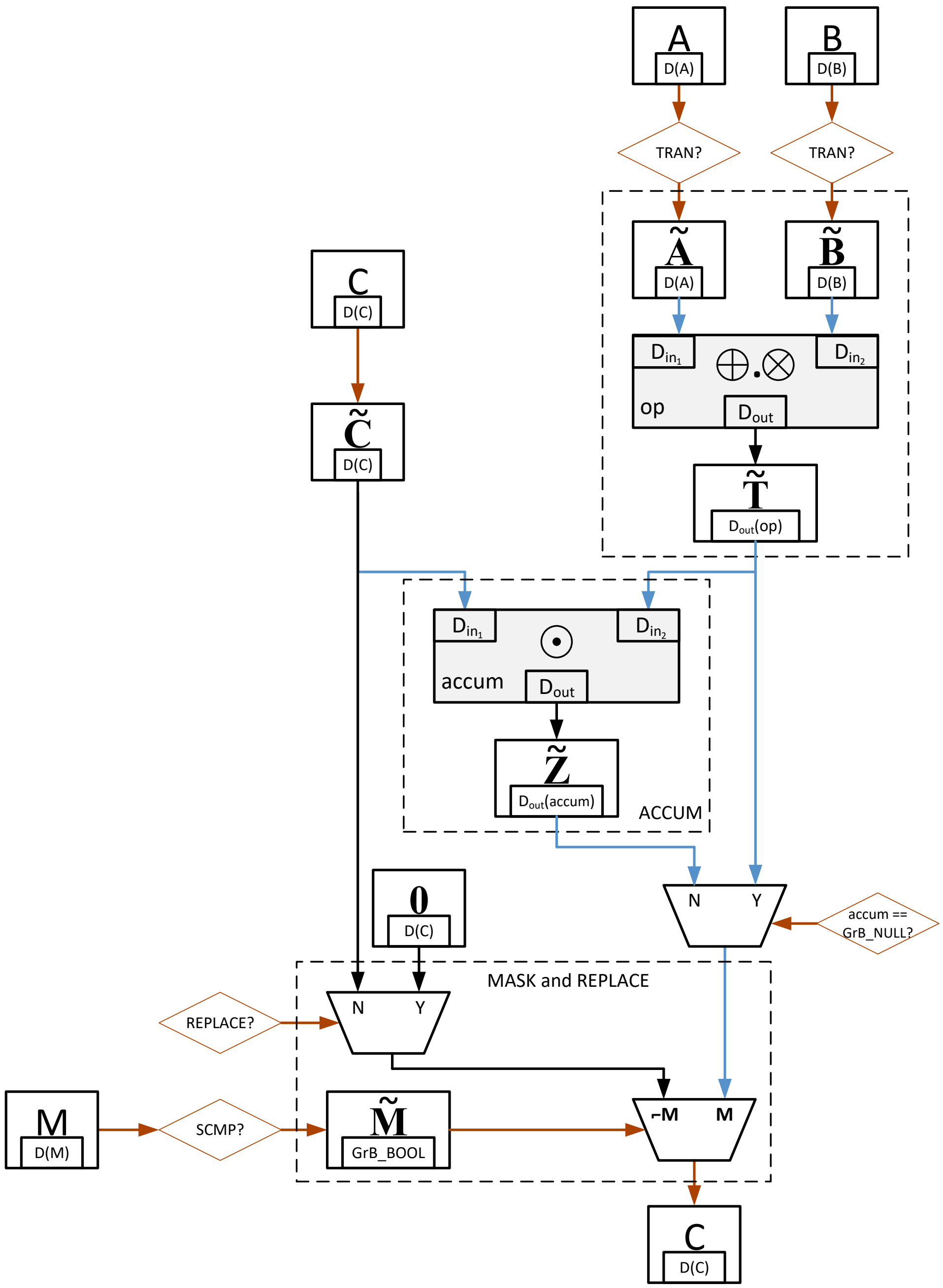

Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the "ACCUM" and "MASK and REPLACE" blocks. Orange arrows denote where "as if copy" takes place (including both collections and descriptor settings). Blue arrows indicate where casting may occur between different domains.

of $\mathbf{C}$ must equal the number of rows of $\mathbf{A}$, the number of columns of $\mathbf{A}$ must match the number of rows of $\mathbf{B}$, and the number of columns of $\mathbf{C}$ must match then number of columns of $\mathbf{B}$. This is the behavior expected given the mathematical definition of the operations.

For most of the GraphBLAS operations involving matrices, an optional descriptor can modify the matrix associated with an input GraphBLAS matrix object. For example, if an input matrix is an argument to a GraphBLAS operation and the associated descriptor indicates the transpose option, then the operation occurs as if on the transposed matrix. In this case, the relationships between the sizes in each dimension shift in the mathematically expected way.

Masks and Structural Complements

When a GraphBLAS operation supports the use of an optional mask, that mask is specified through a GraphBLAS vector (for one-dimensional masks) or a GraphBLAS matrix (for two-dimensional masks). When a mask is used, it is applied to the result from the operation whereever the mask evaluates to true, and then that result is either assigned to the provided output matrix/vector or, if a binary accumulation operation is provided, the result is accumulated into the corresponding elements of the provided output matrix/vector.

Given a GraphBLAS vector $\mathbf{v} = \langle D, N, \{(i, v_i)\}\rangle$, a one-dimensional mask $\mathbf{m} = \langle N, \{i : \texttt{(bool)}v_i = \textsf{true}\}\rangle$ is derived for use in the operation, where $\texttt{(bool)}v_i$ denotes casting the value v_i to a Boolean value (true or false).

Given a GraphBLAS matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\}\rangle$, a two-dimensional mask $\mathbf{M} = \langle M, N, \{(i, j) : \texttt{(bool)}A_{ij} = \textsf{true}\}\rangle$ is derived for use in the operation, where $\texttt{(bool)}A_{ij}$ denotes casting the value A_{ij} to a Boolean value (true or false).

In both the one- and two-dimensional cases, the mask may go through a structural complement operation (§ 3.6) as specified in the descriptor, before a final mask is generated for use in the operation.

When the descriptor of an operation with a mask has specified that the GrB_REPLACE value is to be applied to the output (GrB_OUTP), then anywhere the mask is not true, the corresponding location in the output is cleared.

Invalid and uninitialized objects

Upon entering a GraphBLAS operation, the first step is a check that all objects are valid and initialized. (Optional parameters can be set to GrB_NULL, which always counts as a valid object.) An invalid object is one that could not be computed due to some previous execution error. An unitialized ojbect is one that has not yet been created by a corresponding new or dup method. Appropriate error codes are returned if an object is not initialized (GrB_UNINITIALIZED_OBJECT) or invalid (GrB_INVALID_OBJECT).

To support the detection of as many cases of uninitialized objects as possible, it is strongly recommended to initialize all GraphBLAS objects to the predefined value GrB_INVALID_HANDLE at the point of their declaration, as shown in the following examples:

```
        GrB_Type        type = GrB_INVALID_HANDLE;
        GrB_Semiring    semiring = GrB_INVALID_HANDLE;
        GrB_Matrix      matrix = GrB_INVALID_HANDLE;
```

Compliance

We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations. That is, for each operation we give a recipe for producing its outcome. It should be understood that any implementation that produces the same outcome, and follows the GraphBLAS execution model (§ 2.8) and error model (§ 2.9), is a conforming implementation.

4.3.1 mxm: Matrix-matrix multiply

Multiplies a matrix with another matrix on a semiring. The result is a matrix.

C Syntax

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

Parameters

- C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the matrix product. On output, the matrix holds the results of the operation.
- Mask (IN) An optional "write" mask that controls which results from this operation are stored into the output matrix C. The mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default matrix is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.
- accum (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.
- op (IN) The semiring used in the matrix-matrix multiply.
- A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the multiplication.

B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the multiplication.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_SCMP	Use the structural complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the corresponding domains of the semiring or accumulation operator, or the mask's domain is not compatible with bool.

Description

GrB_mxm computes the matrix product $\mathsf{C} = \mathsf{A} \otimes . \oplus \mathsf{B}$ or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C} = \mathsf{C} \odot (\mathsf{A} \otimes . \oplus \mathsf{B})$ (where matrices A and B can be optionally transposed). Logically, this operation occurs in three steps:

Setup The internal matrices and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to four argument matrices are used in the GrB_mxm operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij})\}\rangle$
2. $\mathsf{Mask} = \langle \mathbf{D}(\mathsf{Mask}), \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \mathbf{L}(\mathsf{Mask}) = \{(i, j, M_{ij})\}\rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\}\rangle$
4. $\mathsf{B} = \langle \mathbf{D}(\mathsf{B}), \mathbf{nrows}(\mathsf{B}), \mathbf{ncols}(\mathsf{B}), \mathbf{L}(\mathsf{B}) = \{(i, j, B_{ij})\}\rangle$

The argument matrices, the semiring, and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{op})$ of the semiring.
3. $\mathbf{D}(\mathsf{B})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{op})$ of the semiring.
4. $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$ of the semiring.
5. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}_{out}(\mathsf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_mxm ends and the domain mismatch error listed above is returned.

From the argument matrices, the internal matrices and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Matrix $\widetilde{\mathbf{C}} \leftarrow \mathsf{C}$.
2. Two-dimensional mask, $\widetilde{\mathbf{M}}$, is computed from argument Mask as follows:
   (a) If $\mathsf{Mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathsf{C}), 0 \leq j < \mathbf{ncols}(\mathsf{C})\}\rangle$.
   (b) Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathsf{Mask}) \wedge (\mathsf{bool})\mathsf{Mask}(i, j) = \mathsf{true}\}\rangle$.

   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}$.

3. Matrix $\widetilde{\mathbf{A}} \leftarrow$ desc[GrB_INP0].GrB_TRAN ? A^T : A.
4. Matrix $\widetilde{\mathbf{B}} \leftarrow$ desc[GrB_INP1].GrB_TRAN ? B^T : B.

The internal matrices and masks are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.
4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{B}})$.
5. $\mathbf{ncols}(\widetilde{\mathbf{A}}) = \mathbf{nrows}(\widetilde{\mathbf{B}})$.

If any compatibility rule above is violated, execution of GrB_mxm ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the matrix multiplication and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\widetilde{\mathbf{T}}$: The matrix holding the product of matrices $\widetilde{\mathbf{A}}$ and $\widetilde{\mathbf{B}}$.
- $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

The intermediate matrix $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{B}}), \{(i, j, T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:, j)) \neq \emptyset\}\rangle$ is created. The value of each of its elements is computed by

$$T_{ij} = \bigoplus_{k \in \mathbf{ind}(\widetilde{\mathbf{A}}(i,:)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:,j))} (\widetilde{\mathbf{A}}(i, k) \otimes \widetilde{\mathbf{B}}(k, j)),$$

where $\oplus$ and $\otimes$ are the additive and multiplicative operators of semiring op, respectively.

The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

  $$\widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

  $$Z_{ij} = \widetilde{\mathbf{C}}(i, j) \odot \widetilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}})),$$

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

$$Z_{ij} = \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up matrix $\widetilde{\mathbf{Z}}$ are written into the final result matrix C, using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in C on input to this operation are deleted and the content of the new output matrix, C, is defined as,

$$\mathbf{L}(\mathsf{C}) = \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{Z}}$ indicated by the mask are copied into the result matrix, C, and elements of C that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{C}) = \{(i,j,C_{ij}) : (i,j) \in (\mathbf{ind}(\mathsf{C}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{M}}))\} \cup \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.2 vxm: Vector-matrix multiply

Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

C Syntax

```
GrB_Info GrB_vxm(GrB_Vector             w,
                 const GrB_Vector       mask,
                 const GrB_BinaryOp     accum,
                 const GrB_Semiring     op,
                 const GrB_Vector       u,
                 const GrB_Matrix       A,
                 const GrB_Descriptor   desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the vector-matrix product. On output, this vector holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w and the domain of the mask vector must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) Semiring used in the vector-matrix multiply.

u (IN) The GraphBLAS vector holding the values for the left-hand vector in the multiplication.

A (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the multiplication.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for matrix or vector parameters).

GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with the corresponding domains of the semiring or accumulation operator, or the mask's domain is not compatible with bool.

Description

GrB_vxm computes the vector-matrix product $\mathsf{w}^T = \mathsf{u}^T \otimes . \oplus \mathsf{A}$, or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{w}^T = \mathsf{w}^T \odot \left(\mathsf{u}^T \otimes . \oplus \mathsf{A}\right)$ (where matrix A can be optionally transposed). Logically, this operation occurs in three steps:

Setup The internal vectors, matrices and mask used in the computation are formed and their domains/dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to four argument vectors or matrices are used in the GrB_vxm operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)
3. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i, u_i)\} \rangle$
4. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\} \rangle$

The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{op})$ of the semiring.
3. $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{op})$ of the semiring.
4. $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$ of the semiring.

5. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}_{out}(\mathsf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_vxm ends and the domain mismatch error listed above is returned.

From the argument vectors and matrices, the internal matrices and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.
2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:
   (a) If $\mathsf{mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i,\ \forall\ i : 0 \leq i < \mathbf{size}(\mathsf{w})\}\rangle$.
   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\}\rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg\widetilde{\mathbf{m}}$.
3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.
4. Matrix $\widetilde{\mathbf{A}} \leftarrow$ desc[GrB_INP1].GrB_TRAN ? $\mathsf{A}^T : \mathsf{A}$.

The internal matrices and masks are checked for shape compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}})$.
2. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.
3. $\mathbf{size}(\widetilde{\mathbf{u}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

If any compatibility rule above is violated, execution of GrB_vxm ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the vector-matrix multiplication and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the product of vector $\widetilde{\mathbf{u}}^T$ and matrix $\widetilde{\mathbf{A}}$.
- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector $\widetilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \{(j, t_j) : \mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{A}}(:, j)) \neq \emptyset\}\rangle$ is created. The value of each of its elements is computed by

$$t_j = \bigoplus_{k \in \mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{A}}(:,j))} (\widetilde{\mathbf{u}}(k) \otimes \widetilde{\mathbf{A}}(k, j)),$$

where $\oplus$ and $\otimes$ are the additive and multiplicative operators of semiring op, respectively.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

  $$\widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i)\ \forall\ i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

  $$z_i = \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i),\ \text{if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})),$$

  $$z_i = \widetilde{\mathbf{w}}(i),\ \text{if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  $$z_i = \widetilde{\mathbf{t}}(i),\ \text{if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\widetilde{\mathbf{z}}$ are written into the final result vector w, using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w, is defined as,

  $$\mathbf{L}(\mathsf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, w, and elements of w that fall outside the set indicated by the mask are unchanged:

  $$\mathbf{L}(\mathsf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathsf{w}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.3 mxv: Matrix-vector multiply

Multiplies a matrix by a vector on a semiring. The result is a vector.

C Syntax

```
GrB_Info GrB_mxv(GrB_Vector             w,
                 const GrB_Vector       mask,
                 const GrB_BinaryOp     accum,
                 const GrB_Semiring     op,
                 const GrB_Matrix       A,
                 const GrB_Vector       u,
                 const GrB_Descriptor   desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the matrix-vector product. On output, this vector holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w and the domain of the mask vector must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) Semiring used in the vector-matrix multiply.

A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the multiplication.

u (IN) The GraphBLAS vector holding the values for the right-hand vector in the multiplication.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for matrix or vector parameters).

GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with the corresponding domains of the semiring or accumulation operator, or the mask's domain is not compatible with bool.

Description

GrB_mxv computes the matrix-vector product $\mathsf{w} = \mathsf{A} \otimes . \oplus \mathsf{u}$, or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{w} = \mathsf{w} \odot (\mathsf{A} \otimes . \oplus \mathsf{u})$ (where matrix A can be optionally transposed). Logically, this operation occurs in three steps:

Setup The internal vectors, matrices and mask used in the computation are formed and their domains/dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to four argument vectors or matrices are used in the GrB_mxv operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\} \rangle$
4. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i, u_i)\} \rangle$

The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{op})$ of the semiring.
3. $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{op})$ of the semiring.
4. $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$ of the semiring.
5. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}_{out}(\mathsf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_mxv ends and the domain mismatch error listed above is returned.

From the argument vectors and matrices, the internal matrices and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.
2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:
   (a) If $\mathsf{mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i,\ \forall\ i : 0 \leq i < \mathbf{size}(\mathsf{w})\} \rangle$.
   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\} \rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg\widetilde{\mathbf{m}}$.
3. Matrix $\widetilde{\mathbf{A}} \leftarrow \mathsf{desc[GrB\_INP0].GrB\_TRAN}\ ?\ \mathsf{A}^T : \mathsf{A}$.
4. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.

The internal matrices and masks are checked for shape compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}})$.
2. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.
3. $\mathbf{size}(\widetilde{\mathbf{u}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.

If any compatibility rule above is violated, execution of GrB_mxv ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the matrix-vector multiplication and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the product of matrix $\widetilde{\mathbf{A}}$ and vector $\widetilde{\mathbf{u}}$.
- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector $\widetilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(i, t_i) : \mathbf{ind}(\widetilde{\mathbf{A}}(i,:)) \cap \mathbf{ind}(\widetilde{\mathbf{u}}) \neq \emptyset\}\rangle$ is created. The value of each of its elements is computed by

$$t_i = \bigoplus_{k \in \mathbf{ind}(\widetilde{\mathbf{A}}(i,:)) \cap \mathbf{ind}(\widetilde{\mathbf{u}})} (\widetilde{\mathbf{A}}(i,k) \otimes \widetilde{\mathbf{u}}(k)),$$

where $\oplus$ and $\otimes$ are the additive and multiplicative operators of semiring op, respectively.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

  $$\widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \ \forall \ i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

  $$z_i = \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})),$$

  $$z_i = \widetilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  $$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\widetilde{\mathbf{z}}$ are written into the final result vector w, using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w, is defined as,

  $$\mathbf{L}(\mathsf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, w, and elements of w that fall outside the set indicated by the mask are unchanged:

  $$\mathbf{L}(\mathsf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathsf{w}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.4 eWiseMult: Element-wise multiplication

Note: The difference between eWiseAdd and eWiseMult is not about the element-wise operation but how the index sets are treated. eWiseAdd returns an object whose indices are the "union" of the indices of the inputs whereas eWiseMult returns an object whose indices are the "intersection" of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on the set of values from the resulting index set.

4.3.4.1 eWiseMult: Vector variant

Perform element-wise (general) multiplication on the intersection of elements of two vectors, producing a third vector as result.

C Syntax

```
GrB_Info GrB_eWiseMult(GrB_Vector             w,
                       const GrB_Vector       mask,
                       const GrB_BinaryOp     accum,
                       const GrB_Semiring     op,
                       const GrB_Vector       u,
                       const GrB_Vector       v,
                       const GrB_Descriptor   desc);

GrB_Info GrB_eWiseMult(GrB_Vector             w,
                       const GrB_Vector       mask,
                       const GrB_BinaryOp     accum,
                       const GrB_Monoid       op,
                       const GrB_Vector       u,
                       const GrB_Vector       v,
                       const GrB_Descriptor   desc);

GrB_Info GrB_eWiseMult(GrB_Vector             w,
                       const GrB_Vector       mask,
                       const GrB_BinaryOp     accum,
                       const GrB_BinaryOp     op,
                       const GrB_Vector       u,
                       const GrB_Vector       v,
                       const GrB_Descriptor   desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the element-wise operation. On output, this vector holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w and the domain of the mask vector must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) The semiring, monoid, or binary operator used in the element-wise "product" operation. Depending on which type is passed, the following defines the binary operator, $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \otimes \rangle$, used:

- BinaryOp: $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \bigodot(\mathsf{op}) \rangle$.
- Monoid: $F_b = \langle \mathbf{D}(\mathsf{op}), \mathbf{D}(\mathsf{op}), \mathbf{D}(\mathsf{op}), \bigodot(\mathsf{op}) \rangle$; the identity element is ignored.
- Semiring: $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \bigotimes(\mathsf{op}) \rangle$; the additive monoid is ignored.

u (IN) The GraphBLAS vector holding the values for the left-hand vector in the operation.

v (IN) The GraphBLAS vector holding the values for the right-hand vector in the operation.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector parameters).

GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the corresponding domains of the binary operator (op) or accumulation operator, or the mask's domain is not compatible with bool.

Description

This variant of GrB_eWiseMult computes the element-wise "product" of two GraphBLAS vectors: $\mathsf{w} = \mathsf{u} \otimes \mathsf{v}$, or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{w} = \mathsf{w} \odot (\mathsf{u} \otimes \mathsf{v})$. Logically, this operation occurs in three steps:

Setup The internal vectors and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to four argument vectors are used in the GrB_eWiseMult operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)
3. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i, u_i)\} \rangle$
4. $\mathsf{v} = \langle \mathbf{D}(\mathsf{v}), \mathbf{size}(\mathsf{v}), \mathbf{L}(\mathsf{v}) = \{(i, v_i)\} \rangle$

The argument vectors, the "product" operator (op), and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{op})$.
3. $\mathbf{D}(\mathsf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{op})$.
4. $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$.

5. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}_{out}(\mathsf{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the domain mismatch error listed above is returned.

From the argument vectors, the internal vectors and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.
2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:
   (a) If mask = GrB_NULL, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i, \ \forall\ i : 0 \leq i < \mathbf{size}(\mathsf{w})\}\rangle$.
   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\}\rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg\widetilde{\mathbf{m}}$.
3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.
4. Vector $\widetilde{\mathbf{v}} \leftarrow \mathsf{v}$.

The internal vectors and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}}) = \mathbf{size}(\widetilde{\mathbf{u}}) = \mathbf{size}(\widetilde{\mathbf{v}})$.

If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the element-wise "product" and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the element-wise "product" of $\widetilde{\mathbf{u}}$ and vector $\widetilde{\mathbf{v}}$.
- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector $\widetilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{size}(\widetilde{\mathbf{u}}), \mathbf{L}(\widetilde{\mathbf{t}}) = \{(i, t_i) : \mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{v}}) \neq \emptyset\}\rangle$ is created. The value of each of its elements is computed by:

$$t_i = (\widetilde{\mathbf{u}}(i) \otimes \widetilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{v}}))$$

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.

- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$\widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \ \forall \ i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})),$$

$$z_i = \widetilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\widetilde{\mathbf{z}}$ are written into the final result vector w, using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w, is defined as,

$$\mathbf{L}(\mathsf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, w, and elements of w that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathsf{w}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.4.2 eWiseMult: Matrix variant

Perform element-wise (general) multiplication on the intersection of elements of two matrices, producing a third matrix as result.

C Syntax

```
GrB_Info GrB_eWiseMult(GrB_Matrix             C,
                       const GrB_Matrix       Mask,
                       const GrB_BinaryOp     accum,
                       const GrB_Semiring     op,
```

```
                                 const GrB_Matrix       A,
                                 const GrB_Matrix       B,
                                 const GrB_Descriptor   desc);

GrB_Info GrB_eWiseMult(GrB_Matrix                       C,
                                 const GrB_Matrix       Mask,
                                 const GrB_BinaryOp     accum,
                                 const GrB_Monoid       op,
                                 const GrB_Matrix       A,
                                 const GrB_Matrix       B,
                                 const GrB_Descriptor   desc);

GrB_Info GrB_eWiseMult(GrB_Matrix                       C,
                                 const GrB_Matrix       Mask,
                                 const GrB_BinaryOp     accum,
                                 const GrB_BinaryOp     op,
                                 const GrB_Matrix       A,
                                 const GrB_Matrix       B,
                                 const GrB_Descriptor   desc);
```

Parameters

C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the element-wise operation. On output, the matrix holds the results of the operation.

Mask (IN) An optional "write" mask that controls which results from this operation are stored into the output matrix C. The mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default matrix is desired (i.e., with correct dimensions end filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) The semiring, monoid, or binary operator used in the element-wise "product" operation. Depending on which type is passed, the following defines the binary operator, $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \otimes \rangle$, used:

BinaryOp: $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \bigodot(\mathsf{op}) \rangle$.

Monoid: $F_b = \langle \mathbf{D}(\mathsf{op}), \mathbf{D}(\mathsf{op}), \mathbf{D}(\mathsf{op}), \bigodot(\mathsf{op}) \rangle$; the identity element is ignored.

Semiring: $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \bigotimes(\mathsf{op}) \rangle$; the additive monoid is ignored.

A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the operation.

B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the operation.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_SCMP	Use the structural complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the corresponding domains of the binary operator (op) or accumulation operator, or the mask's domain is not compatible with bool.

Description

This variant of GrB_eWiseMult computes the element-wise "product" of two GraphBLAS matrices: $\mathsf{C} = \mathsf{A} \otimes \mathsf{B}$, or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C} = \mathsf{C} \odot (\mathsf{A} \otimes \mathsf{B})$. Logically, this operation occurs in three steps:

Setup The internal matrices and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to four argument matrices are used in the GrB_eWiseMult operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij})\} \rangle$
2. $\mathsf{Mask} = \langle \mathbf{D}(\mathsf{Mask}), \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \mathbf{L}(\mathsf{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\} \rangle$
4. $\mathsf{B} = \langle \mathbf{D}(\mathsf{B}), \mathbf{nrows}(\mathsf{B}), \mathbf{ncols}(\mathsf{B}), \mathbf{L}(\mathsf{B}) = \{(i, j, B_{ij})\} \rangle$

The argument matrices, the "product" operator (op), and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{op})$.
3. $\mathbf{D}(\mathsf{B})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{op})$.
4. $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$.
5. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}_{out}(\mathsf{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the domain mismatch error listed above is returned.

From the argument matrices, the internal matrices and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Matrix $\widetilde{\mathbf{C}} \leftarrow \mathsf{C}$.

2. Two-dimensional mask, $\widetilde{\mathbf{M}}$, is computed from argument Mask as follows:

   (a) If $\mathsf{Mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \{(i,j), \forall i,j : 0 \leq i < \mathbf{nrows}(\mathsf{C}), 0 \leq j < \mathbf{ncols}(\mathsf{C})\}\rangle$.

   (b) Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \{(i,j) : (i,j) \in \mathbf{ind}(\mathsf{Mask}) \wedge (\mathsf{bool})\mathsf{Mask}(i,j) = \mathsf{true}\}\rangle$.

   (c) If $\mathsf{desc[GrB\_MASK].GrB\_SCMP}$ is set, then $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}$.

3. Matrix $\widetilde{\mathbf{A}} \leftarrow \mathsf{desc[GrB\_INP0].GrB\_TRAN}\ ?\ \mathsf{A}^T : \mathsf{A}$.

4. Matrix $\widetilde{\mathbf{B}} \leftarrow \mathsf{desc[GrB\_INP1].GrB\_TRAN}\ ?\ \mathsf{B}^T : \mathsf{B}$.

The internal matrices and masks are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}}) = \mathbf{nrows}(\widetilde{\mathbf{A}}) = \mathbf{nrows}(\widetilde{\mathbf{C}})$.

2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}}) = \mathbf{ncols}(\widetilde{\mathbf{A}}) = \mathbf{ncols}(\widetilde{\mathbf{C}})$.

If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the element-wise "product" and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\widetilde{\mathbf{T}}$: The matrix holding the element-wise product of $\widetilde{\mathbf{A}}$ and $\widetilde{\mathbf{B}}$.
- $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

The intermediate matrix $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \{(i,j,T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}}) \neq \emptyset\}\rangle$ is created. The value of each of its elements is computed by

$$T_{ij} = (\widetilde{\mathbf{A}}(i,j) \otimes \widetilde{\mathbf{B}}(i,j)), \forall (i,j) \in \mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}})$$

The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- If $\mathsf{accum} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

  $$\widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i,j,Z_{ij}) \forall (i,j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

  $$Z_{ij} = \widetilde{\mathbf{C}}(i,j) \odot \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}})),$$

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

$$Z_{ij} = \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up matrix $\widetilde{\mathbf{Z}}$ are written into the final result matrix C, using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in C on input to this operation are deleted and the content of the new output matrix, C, is defined as,

$$\mathbf{L}(\mathsf{C}) = \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{Z}}$ indicated by the mask are copied into the result matrix, C, and elements of C that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{C}) = \{(i,j,C_{ij}) : (i,j) \in (\mathbf{ind}(\mathsf{C}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{M}}))\} \cup \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.5 eWiseAdd: Element-wise addition

Note: The difference between eWiseAdd and eWiseMult is not about the element-wise operation but how the index sets are treated. eWiseAdd returns an object whose indices are the "union" of the indices of the inputs whereas eWiseMult returns an object whose indices are the "intersection" of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on the set of values from the resulting index set.

4.3.5.1 eWiseAdd: Vector variant

Perform element-wise (general) addition on the elements of two vectors, producing a third vector as result.

C Syntax

```
GrB_Info GrB_eWiseAdd(GrB_Vector             w,
                      const GrB_Vector       mask,
```

```
                              const GrB_BinaryOp     accum,
                              const GrB_Semiring     op,
                              const GrB_Vector       u,
                              const GrB_Vector       v,
                              const GrB_Descriptor   desc);

GrB_Info GrB_eWiseAdd(GrB_Vector                     w,
                              const GrB_Vector       mask,
                              const GrB_BinaryOp     accum,
                              const GrB_Monoid       op,
                              const GrB_Vector       u,
                              const GrB_Vector       v,
                              const GrB_Descriptor   desc);

GrB_Info GrB_eWiseAdd(GrB_Vector                     w,
                              const GrB_Vector       mask,
                              const GrB_BinaryOp     accum,
                              const GrB_BinaryOp     op,
                              const GrB_Vector       u,
                              const GrB_Vector       v,
                              const GrB_Descriptor   desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the element-wise operation. On output, this vector holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w and the domain of the mask vector must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) The semiring, monoid, or binary operator used in the element-wise "sum" operation. Depending on which type is passed, the following defines the binary operator, $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \oplus \rangle$, used:

BinaryOp: $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \bigodot(\mathsf{op}) \rangle$.

Monoid: $F_b = \langle \mathbf{D}(\mathsf{op}), \mathbf{D}(\mathsf{op}), \mathbf{D}(\mathsf{op}), \bigodot(\mathsf{op}) \rangle$; the identity element is ignored.

Semiring: $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \bigoplus(\mathsf{op})\rangle$; the multiplicative binary op and additive identity are ignored.

u (IN) The GraphBLAS vector holding the values for the left-hand vector in the operation.

v (IN) The GraphBLAS vector holding the values for the right-hand vector in the operation.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector parameters).

GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the corresponding domains of the binary operator (op) or accumulation operator, or the mask's domain is not compatible with bool.

Description

This variant of GrB_eWiseAdd computes the element-wise "sum" of two GraphBLAS vectors: $\mathsf{w} = \mathsf{u} \oplus \mathsf{v}$, or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{w} = \mathsf{w} \odot (\mathsf{u} \oplus \mathsf{v})$. Logically, this operation occurs in three steps:

Setup The internal vectors and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to four argument vectors are used in the GrB_eWiseAdd operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)
3. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i, u_i)\} \rangle$
4. $\mathsf{v} = \langle \mathbf{D}(\mathsf{v}), \mathbf{size}(\mathsf{v}), \mathbf{L}(\mathsf{v}) = \{(i, v_i)\} \rangle$

The argument vectors, the "sum" operator (op), and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{op})$.
3. $\mathbf{D}(\mathsf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{op})$.
4. $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$.
5. $\mathbf{D}(\mathsf{u})$ and $\mathbf{D}(\mathsf{v})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$.
6. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}_{out}(\mathsf{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the domain mismatch error listed above is returned.

From the argument vectors, the internal vectors and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.

2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:

   (a) If mask = GrB_NULL, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i, \ \forall\ i : 0 \leq i < \mathbf{size}(\mathsf{w})\}\rangle$.

   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\}\rangle$.

   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg\widetilde{\mathbf{m}}$.

3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.

4. Vector $\widetilde{\mathbf{v}} \leftarrow \mathsf{v}$.

The internal vectors and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}}) = \mathbf{size}(\widetilde{\mathbf{u}}) = \mathbf{size}(\widetilde{\mathbf{v}})$.

If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the element-wise "sum" and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the element-wise "sum" of $\widetilde{\mathbf{u}}$ and vector $\widetilde{\mathbf{v}}$.

- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector $\widetilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{size}(\widetilde{\mathbf{u}}), \mathbf{L}(\widetilde{\mathbf{t}}) = \{(i, t_i) : \mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{v}}) \neq \emptyset\}\rangle$ is created. The value of each of its elements is computed by:

$$t_i = (\widetilde{\mathbf{u}}(i) \oplus \widetilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{v}}))$$

$$t_i = \widetilde{\mathbf{u}}(i), \forall i \in (\mathbf{ind}(\widetilde{\mathbf{u}}) - (\mathbf{ind}(\widetilde{\mathbf{v}}) \cap \mathbf{ind}(\widetilde{\mathbf{u}})))$$

$$t_i = \widetilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\widetilde{\mathbf{v}}) - (\mathbf{ind}(\widetilde{\mathbf{v}}) \cap \mathbf{ind}(\widetilde{\mathbf{u}})))$$

where the difference operator in the previous expressions refers to set difference.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.

- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$\widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i)\ \forall\ i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i),\ \text{if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})),$$

$$z_i = \widetilde{\mathbf{w}}(i),\ \text{if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i),\ \text{if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\widetilde{\mathbf{z}}$ are written into the final result vector w, using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w, is defined as,

$$\mathbf{L}(\mathsf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, w, and elements of w that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathsf{w}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.5.2 eWiseAdd: Matrix variant

Perform element-wise (general) addition on the elements of two matrices, producing a third matrix as result.

C Syntax

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

GrB_Info GrB_eWiseAdd(GrB_Matrix              C,
```

```
                             const GrB_Matrix       Mask,
                             const GrB_BinaryOp     accum,
                             const GrB_Monoid       op,
                             const GrB_Matrix       A,
                             const GrB_Matrix       B,
                             const GrB_Descriptor   desc);

GrB_Info GrB_eWiseAdd(GrB_Matrix                    C,
                             const GrB_Matrix       Mask,
                             const GrB_BinaryOp     accum,
                             const GrB_BinaryOp     op,
                             const GrB_Matrix       A,
                             const GrB_Matrix       B,
                             const GrB_Descriptor   desc);
```

Parameters

- **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the element-wise operation. On output, the matrix holds the results of the operation.

- **Mask** (IN) An optional "write" mask that controls which results from this operation are stored into the output matrix C. The mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default matrix is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

- **accum** (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

- **op** (IN) The semiring, monoid, or binary operator used in the element-wise "sum" operation. Depending on which type is passed, the following defines the binary operator, $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \oplus \rangle$, used:

  BinaryOp: $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \bigodot(\mathsf{op}) \rangle$.

  Monoid: $F_b = \langle \mathbf{D}(\mathsf{op}), \mathbf{D}(\mathsf{op}), \mathbf{D}(\mathsf{op}), \bigodot(\mathsf{op}) \rangle$; the identity element is ignored.

  Semiring: $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \bigoplus(\mathsf{op}) \rangle$; the multiplicative binary op and additive identity are ignored.

- **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the operation.

- **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the operation.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_SCMP	Use the structural complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the corresponding domains of the binary operator (op) or accumulation operator, or the mask's domain is not compatible with bool.

Description

This variant of GrB_eWiseAdd computes the element-wise "sum" of two GraphBLAS matrices: $\mathsf{C} = \mathsf{A} \oplus \mathsf{B}$, or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C} = \mathsf{C} \odot (\mathsf{A} \oplus \mathsf{B})$. Logically, this operation occurs in three steps:

Setup The internal matrices and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to four argument matrices are used in the GrB_eWiseMult operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij})\}\rangle$
2. $\mathsf{Mask} = \langle \mathbf{D}(\mathsf{Mask}), \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \mathbf{L}(\mathsf{Mask}) = \{(i, j, M_{ij})\}\rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\}\rangle$
4. $\mathsf{B} = \langle \mathbf{D}(\mathsf{B}), \mathbf{nrows}(\mathsf{B}), \mathbf{ncols}(\mathsf{B}), \mathbf{L}(\mathsf{B}) = \{(i, j, B_{ij})\}\rangle$

The argument matrices, the "sum" operator (op), and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{op})$.
3. $\mathbf{D}(\mathsf{B})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{op})$.
4. $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$.
5. $\mathbf{D}(\mathsf{A})$ and $\mathbf{D}(\mathsf{B})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$.
6. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}_{out}(\mathsf{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the domain mismatch error listed above is returned.

From the argument matrices, the internal matrices and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Matrix $\widetilde{\mathbf{C}} \leftarrow \mathsf{C}$.
2. Two-dimensional mask, $\widetilde{\mathbf{M}}$, is computed from argument Mask as follows:
   (a) If $\mathsf{Mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathsf{C}), 0 \leq j < \mathbf{ncols}(\mathsf{C})\}\rangle$.
   (b) Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathsf{Mask}) \wedge (\mathsf{bool})\mathsf{Mask}(i, j) = \mathsf{true}\}\rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}$.

3. Matrix $\widetilde{\mathbf{A}} \leftarrow$ desc[GrB_INP0].GrB_TRAN ? A^T : A.
4. Matrix $\widetilde{\mathbf{B}} \leftarrow$ desc[GrB_INP1].GrB_TRAN ? B^T : B.

The internal matrices and masks are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}}) = \mathbf{nrows}(\widetilde{\mathbf{A}}) = \mathbf{nrows}(\widetilde{\mathbf{C}})$.
2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}}) = \mathbf{ncols}(\widetilde{\mathbf{A}}) = \mathbf{ncols}(\widetilde{\mathbf{C}})$.

If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the element-wise "sum" and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\widetilde{\mathbf{T}}$: The matrix holding the element-wise sum of $\widetilde{\mathbf{A}}$ and $\widetilde{\mathbf{B}}$.
- $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

The intermediate matrix $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}}) \neq \emptyset\}\rangle$ is created. The value of each of its elements is computed by

$$T_{ij} = (\widetilde{\mathbf{A}}(i, j) \oplus \widetilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}})$$

$$T_{ij} = \widetilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{A}}) - (\mathbf{ind}(\widetilde{\mathbf{B}}) \cap \mathbf{ind}(\widetilde{\mathbf{A}})))$$

$$T_{ij} = \widetilde{\mathbf{B}}(i.j), \forall (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{B}}) - (\mathbf{ind}(\widetilde{\mathbf{B}}) \cap \mathbf{ind}(\widetilde{\mathbf{A}})))$$

where the difference operator in the previous expressions refers to set difference.

The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

  $$\widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

  $$Z_{ij} = \widetilde{\mathbf{C}}(i, j) \odot \widetilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}})),$$

  $$Z_{ij} = \widetilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

  $$Z_{ij} = \widetilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up matrix $\widetilde{\mathbf{Z}}$ are written into the final result matrix C, using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in C on input to this operation are deleted and the content of the new output matrix, C, is defined as,

$$\mathbf{L}(\mathsf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{Z}}$ indicated by the mask are copied into the result matrix, C, and elements of C that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathsf{C}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.6 extract: Selecting Sub-Graphs

Extract a subset of a matrix or vector.

4.3.6.1 extract: Standard vector variant

Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector whose size is equal to the number of indices.

C Syntax

```
GrB_Info GrB_extract(GrB_Vector              w,
                     const GrB_Vector        mask,
                     const GrB_BinaryOp      accum,
                     const GrB_Vector        u,
                     const GrB_Index        *indices,
                     GrB_Index               nindices,
                     const GrB_Descriptor    desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the extract operation. On output, this vector holds the results of the operation.

mask (IN) An optional “write” mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w and the domain of the mask vector must be of type bool or any of the predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

u (IN) The GraphBLAS vector from which the subset is extracted.

indices (IN) Pointer to the ordered set (array) of indices corresponding to the locations of elements from u that are extracted. If all elements of u are to be extracted in order from 0 to nindices $-$ 1, then GrB_ALL should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

nindices (IN) The number of values in indices array. Must be equal to **size**(w).

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector parameters).

GrB_INDEX_OUT_OF_BOUNDS	A value in indices is greater than or equal to **size**(u). In non-blocking mode, this error can be deferred.
GrB_DIMENSION_MISMATCH	mask and w dimensions are incompatible, or nindices $\neq$ **size**(w).
GrB_DOMAIN_MISMATCH	The domains of the various vectors are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.
GrB_NULL_POINTER	Argument row_indices is a NULL pointer.

Description

This variant of GrB_extract computes the result of extracting a subset of locations from a GraphBLAS vector in a specific order: $\mathsf{w} = \mathsf{u}(\mathsf{indices})$; or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{w} = \mathsf{w} \odot \mathsf{u}(\mathsf{indices})$. More explicitly:

$$\begin{aligned} \mathsf{w}(i) &= \quad\quad\quad \mathsf{u}(\mathsf{indices}[i]),\ \forall\ i:\ 0 \leq i < \mathsf{nindices}, \quad \text{or} \\ \mathsf{w}(i) &= \mathsf{w}(i) \odot \mathsf{u}(\mathsf{indices}[i]),\ \forall\ i:\ 0 \leq i < \mathsf{nindices} \end{aligned}$$

Logically, this operation occurs in three steps:

Setup The internal vectors and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to three argument vectors are used in this GrB_extract operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)
3. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i, u_i)\} \rangle$

The argument vectors and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}(\mathsf{u})$.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch error listed above is returned.

From the arguments, the internal vectors, mask, and index array used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.
2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:
   (a) If $\mathsf{mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i,\ \forall\ i : 0 \leq i < \mathbf{size}(\mathsf{w})\}\rangle$.
   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\}\rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg\widetilde{\mathbf{m}}$.
3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.
4. The internal index array, $\widetilde{\boldsymbol{I}}$, is computed from argument indices as follows:
   (a) If $\mathsf{indices} = \mathsf{GrB\_ALL}$, then $\widetilde{\boldsymbol{I}}[i] = i,\ \forall\ i : 0 \leq i < \mathsf{nindices}$.
   (b) Otherwise, $\widetilde{\boldsymbol{I}}[i] = \mathsf{indices}[i],\ \forall\ i : 0 \leq i < \mathsf{nindices}$.

The internal vectors and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}})$
2. $\mathsf{nindices} = \mathbf{size}(\widetilde{\mathbf{w}})$.

If any compatibility rule above is violated, execution of GrB_extract ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the extract and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the extraction from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{w}}$.
- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$\widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}(\widetilde{\boldsymbol{I}}[i]))\ \forall\ i, 0 \leq i < \mathsf{nindices} : \widetilde{\boldsymbol{I}}[i] \in \mathbf{ind}(\widetilde{\mathbf{u}})\}\rangle.$$

At this point, if any value in $\widetilde{\boldsymbol{I}}$ is not in the valid range of indices for vector $\widetilde{\mathbf{u}}$, the execution of GrB_extract ends and the index-out-of-bounds error listed above is generated. In GrB_NONBLOCKING mode, the error can be deferred until a sequence-terminating GrB_wait() is called. Regardless, the result vector, w, is invalid from this point forward in the sequence.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$\widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i)\ \forall\ i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i),\ \text{if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})),$$

$$z_i = \widetilde{\mathbf{w}}(i),\ \text{if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i),\ \text{if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\widetilde{\mathbf{z}}$ are written into the final result vector w, using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w, is defined as,

$$\mathbf{L}(\mathsf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, w, and elements of w that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathsf{w}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.6.2 extract: Standard matrix variant

Extract a sub-matrix from a larger matrix as specified by a set of row indices and a set of column indices. The result is a matrix whose size is equal to size of the sets of indices.

C Syntax

```
GrB_Info GrB_extract(GrB_Matrix            C,
                     const GrB_Matrix      Mask,
                     const GrB_BinaryOp    accum,
                     const GrB_Matrix      A,
                     const GrB_Index      *row_indices,
                     GrB_Index             nrows,
                     const GrB_Index      *col_indices,
                     GrB_Index             ncols,
                     const GrB_Descriptor  desc);
```

Parameters

- **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the extract operation. On output, the matrix holds the results of the operation.
- **Mask** (IN) An optional "write" mask that controls which results from this operation are stored into the output matrix C. The mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default matrix is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.
- **accum** (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.
- **A** (IN) The GraphBLAS matrix from which the subset is extracted.
- **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of A from which elements are extracted. If elements in all rows of A are to be extracted in order, GrB_ALL should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.
- **nrows** (IN) The number of values in the row_indices array. Must be equal to **nrows**(C).
- **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of A from which elements are extracted. If elements in all columns of A are to be extracted in order, then GrB_ALL should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.
- **ncols** (IN) The number of values in the col_indices array. Must be equal to **ncols**(C).
- **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_SCMP	Use the structural complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to **nrows**(A), or a value in col_indices is greater than or equal to **ncols**(A). In non-blocking mode, this error can be deferred.

GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrows $\neq$ **nrows**(C), or ncols $\neq$ **ncols**(C).

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.

GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices is a NULL pointer, or both.

Description

This variant of GrB_extract computes the result of extracting a subset of locations from specified rows and columns of a GraphBLAS matrix in a specific order: C = A(row_indices, col_indices); or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C} = \mathsf{C} \odot \mathsf{A}(\mathsf{row\_indices}, \mathsf{col\_indices})$. More explicitly (not accounting for an optional transpose of A):

$$
\begin{aligned}
\mathsf{C}(i,j) &= \phantom{\mathsf{C}(i,j) \odot{}} \mathsf{A}(\mathsf{row\_indices}[i], \mathsf{col\_indices}[j]) \ \forall\ i,j \ : \ 0 \le i < \mathsf{nrows},\ 0 \le j < \mathsf{ncols}, \text{ or} \\
\mathsf{C}(i,j) &= \mathsf{C}(i,j) \odot \mathsf{A}(\mathsf{row\_indices}[i], \mathsf{col\_indices}[j]) \ \forall\ i,j \ : \ 0 \le i < \mathsf{nrows},\ 0 \le j < \mathsf{ncols}
\end{aligned}
$$

Logically, this operation occurs in three steps:

Setup The internal matrices and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to three argument matrices are used in the GrB_extract operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i,j,C_{ij})\} \rangle$
2. $\mathsf{Mask} = \langle \mathbf{D}(\mathsf{Mask}), \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \mathbf{L}(\mathsf{Mask}) = \{(i,j,M_{ij})\} \rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i,j,A_{ij})\} \rangle$

The argument matrices and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}(\mathsf{A})$.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch error listed above is returned.

From the arguments, the internal matrices, mask, and index arrays used in the computation are formed ($\leftarrow$ denotes copy):

1. Matrix $\widetilde{\mathbf{C}} \leftarrow \mathsf{C}$.
2. Two-dimensional mask, $\widetilde{\mathbf{M}}$, is computed from argument Mask as follows:
   (a) If $\mathsf{Mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \{(i,j), \forall i,j : 0 \le i < \mathbf{nrows}(\mathsf{C}), 0 \le j < \mathbf{ncols}(\mathsf{C})\} \rangle$.

(b) Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \{(i,j) : (i,j) \in \mathbf{ind}(\mathsf{Mask}) \wedge (\mathsf{bool})\mathsf{Mask}(i,j) = \mathsf{true}\}\rangle$.

(c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}$.

3. Matrix $\widetilde{\mathbf{A}} \leftarrow$ desc[GrB_INP0].GrB_TRAN ? A^T : A.

4. The internal row index array, $\widetilde{\boldsymbol{I}}$, is computed from argument row_indices as follows:

   (a) If row_indices = GrB_ALL, then $\widetilde{\boldsymbol{I}}[i] = i, \forall i : 0 \leq i < \mathsf{nrows}$.

   (b) Otherwise, $\widetilde{\boldsymbol{I}}[i] = \mathsf{row\_indices}[i], \forall i : 0 \leq i < \mathsf{nrows}$.

5. The internal column index array, $\widetilde{\boldsymbol{J}}$, is computed from argument col_indices as follows:

   (a) If col_indices = GrB_ALL, then $\widetilde{\boldsymbol{J}}[j] = j, \forall j : 0 \leq j < \mathsf{ncols}$.

   (b) Otherwise, $\widetilde{\boldsymbol{J}}[j] = \mathsf{col\_indices}[j], \forall j : 0 \leq j < \mathsf{ncols}$.

The internal matrices and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.

2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.

3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathsf{nrows}$.

4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathsf{ncols}$.

If any compatibility rule above is violated, execution of GrB_extract ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the extract and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\widetilde{\mathbf{T}}$: The matrix holding the extraction from $\widetilde{\mathbf{A}}$.

- $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

The intermediate matrix, $\widetilde{\mathbf{T}}$, is created as follows:

$$\begin{aligned}\widetilde{\mathbf{T}} = \langle &\mathbf{D}(\mathsf{A}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}),\\ &\{(i, j, \widetilde{\mathbf{A}}(\widetilde{\boldsymbol{I}}[i], \widetilde{\boldsymbol{J}}[j]))\ \forall\ (i,j),\ 0 \leq i < \mathsf{nrows},\ 0 \leq j < \mathsf{ncols} : (\widetilde{\boldsymbol{I}}[i], \widetilde{\boldsymbol{J}}[j]) \in \mathbf{ind}(\widetilde{\mathbf{A}})\}\rangle.\end{aligned}$$

At this point, if any value in the $\widetilde{\boldsymbol{I}}$ array is not in the range $[0,\ \mathbf{nrows}(\widetilde{\mathbf{A}}))$ or any value in the $\widetilde{\boldsymbol{J}}$ array is not in the range $[0,\ \mathbf{ncols}(\widetilde{\mathbf{A}}))$, the execution of GrB_extract ends and the index out-of-bounds error listed above is generated. In GrB_NONBLOCKING mode, the error can be deferred until a sequence-terminating GrB_wait() is called. Regardless, the result matrix C is invalid from this point forward in the sequence.

The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

  $$\widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i,j,Z_{ij}) \forall (i,j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

  The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

  $$Z_{ij} = \widetilde{\mathbf{C}}(i,j) \odot \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}})),$$

  $$Z_{ij} = \widetilde{\mathbf{C}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

  $$Z_{ij} = \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up matrix $\widetilde{\mathbf{Z}}$ are written into the final result matrix C, using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in C on input to this operation are deleted and the content of the new output matrix, C, is defined as,

  $$\mathbf{L}(\mathsf{C}) = \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{Z}}$ indicated by the mask are copied into the result matrix, C, and elements of C that fall outside the set indicated by the mask are unchanged:

  $$\mathbf{L}(\mathsf{C}) = \{(i,j,C_{ij}) : (i,j) \in (\mathbf{ind}(\mathsf{C}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{M}}))\} \cup \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.6.3 extract: Column (and row) variant

Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the source matrix, elements of an arbitrary row of the matrix can be extracted with this function as well.

C Syntax

```
GrB_Info GrB_extract(GrB_Vector             w,
                     const GrB_Vector        mask,
                     const GrB_BinaryOp      accum,
                     const GrB_Matrix        A,
                     const GrB_Index        *row_indices,
                     GrB_Index               nrows,
                     GrB_Index               col_index,
                     const GrB_Descriptor    desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the extract operation. On output, this vector holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w and the domain of the mask vector must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

A (IN) The GraphBLAS matrix from which the column subset is extracted.

row_indices (IN) Pointer to the ordered set (array) of indices corresponding to the locations within the specified column of A from which elements are extracted. If elements in all rows of A are to be extracted in order, GrB_ALL should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

nrows (IN) The number of indices in the row_indices array. Must be equal to **size**(w).

col_index (IN) The index of the column of A from which to extract values. It must be in the range [0, **ncols**(A)).

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

Return Values

- **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.
- **GrB_PANIC** Unknown internal error.
- **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
- **GrB_OUT_OF_MEMORY** Not enough memory available for operation.
- **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector or matrix parameters).
- **GrB_INVALID_INDEX** col_index is outside the allowable range (i.e., greater than **ncols**(A)).
- **GrB_INDEX_OUT_OF_BOUNDS** A value in row_indices is greater than or equal to **nrows**(A). In non-blocking mode, this error can be deferred.
- **GrB_DIMENSION_MISMATCH** mask and w dimensions are incompatible, or nrows $\neq$ **size**(w).
- **GrB_DOMAIN_MISMATCH** The domains of the vector or matrix are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.
- **GrB_NULL_POINTER** Argument row_indices is a NULL pointer.

Description

This variant of GrB_extract computes the result of extracting a subset of locations (in a specific order) from a specified column of a GraphBLAS matrix: $\mathsf{w} = \mathsf{A}(:, \mathsf{col\_index})(\mathsf{row\_indices})$; or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{w} = \mathsf{w} \odot \mathsf{A}(:, \mathsf{col\_index})(\mathsf{row\_indices})$. More explicitly:

$$\begin{aligned} \mathsf{w}(i) &= \qquad\quad \mathsf{A}(\mathsf{row\_indices}[i], \mathsf{col\_index}) \ \forall \ i: \ 0 \leq i < \mathsf{nrows}, \quad \text{or} \\ \mathsf{w}(i) &= \mathsf{w}(i) \odot \mathsf{A}(\mathsf{row\_indices}[i], \mathsf{col\_index}) \ \forall \ i: \ 0 \leq i < \mathsf{nrows} \end{aligned}$$

Logically, this operation occurs in three steps:

Setup The internal matrices, vectors, and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to three argument vectors and matrices are used in this GrB_extract operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\} \rangle$

The argument vectors, matrix and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}(\mathsf{A})$.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_extract ends and the domain mismatch error listed above is returned.

From the arguments, the internal vector, matrix, mask, and index array used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.
2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:
   (a) If $\mathsf{mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i, \ \forall \ i : 0 \leq i < \mathbf{size}(\mathsf{w})\} \rangle$.
   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\} \rangle$.

(c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg\widetilde{\mathbf{m}}$.

3. Matrix $\widetilde{\mathbf{A}} \leftarrow$ desc[GrB_INP0].GrB_TRAN ? A^T : A.

4. The internal row index array, $\widetilde{\boldsymbol{I}}$, is computed from argument row_indices as follows:

   (a) If indices = GrB_ALL, then $\widetilde{\boldsymbol{I}}[i] = i, \ \forall \ i : 0 \leq i < \mathsf{nrows}$.

   (b) Otherwise, $\widetilde{\boldsymbol{I}}[i] = \mathsf{indices}[i], \ \forall \ i : 0 \leq i < \mathsf{nrows}$.

The internal vector, mask, and index array are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}})$

2. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathsf{nrows}$.

If any compatibility rule above is violated, execution of GrB_extract ends and the dimension mismatch error listed above is returned.

The col_index parameter is checked for a valid value. The following condition must hold:

1. $0 \ \leq \ \mathsf{col\_index} \ < \ \mathbf{ncols}(\mathsf{A})$

If the rule above is violated, execution of GrB_extract ends and the invalid index error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the extract and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the extraction from a column of $\widetilde{\mathbf{A}}$.
- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$\widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathsf{A}), \mathsf{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\boldsymbol{I}}[i], \mathsf{col\_index})) \ \forall \ i, 0 \leq i < \mathsf{nrows} : (\widetilde{\boldsymbol{I}}[i], \mathsf{col\_index}) \in \mathbf{ind}(\widetilde{\mathbf{A}})\}\rangle.$$

At this point, if any value in $\widetilde{\boldsymbol{I}}$ is not in the range $[0, \ \mathbf{nrows}(\widetilde{\mathbf{A}}))$, the execution of GrB_extract ends and the index-out-of-bounds error listed above is generated. In GrB_NONBLOCKING mode, the error can be deferred until a sequence-terminating GrB_wait() is called. Regardless, the result vector, w, is invalid from this point forward in the sequence.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.

- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$\widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \ \forall \ i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})),$$

$$z_i = \widetilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\widetilde{\mathbf{z}}$ are written into the final result vector w, using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w, is defined as,

$$\mathbf{L}(\mathsf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, w, and elements of w that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathsf{w}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.7 assign: Modifying Sub-Graphs

Assign the contents of a subset of a matrix or vector.

4.3.7.1 assign: Standard vector variant

Assign values (and implied zeros) from one GraphBLAS vector to a subset of a vector as specified by a set of indices. The size of the input vector is the same size as the index array provided.

C Syntax

```
GrB_Info GrB_assign(GrB_Vector             w,
                    const GrB_Vector       mask,
                    const GrB_BinaryOp     accum,
                    const GrB_Vector       u,
                    const GrB_Index       *indices,
                    GrB_Index              nindices,
                    const GrB_Descriptor   desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the assign operation. On output, this vector holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w and the domain of the mask vector must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

u (IN) The GraphBLAS vector whose contents are assigned to a subset of w.

indices (IN) Pointer to the ordered set (array) of indices corresponding to the locations in w that are to be assigned. If all elements of w are to be assigned in order from 0 to nindices $-$ 1, then GrB_ALL should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation. If this array contains duplicate values, it implies in assignment of more than one value to the same location which leads to undefined results.

nindices (IN) The number of values in indices array. Must be equal to **size**(u).

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector parameters).

GrB_INDEX_OUT_OF_BOUNDS A value in indices is greater than or equal to **size**(w). In non-blocking mode, this can be reported as an execution error.

GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or nindices $\neq$ **size**(u).

GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.

GrB_NULL_POINTER Argument indices is a NULL pointer.

Description

This variant of GrB_assign computes the result of assigning elements from a source GraphBLAS vector to a destination GraphBLAS vector in a specific order: w(indices) = u; or, if an optional binary accumulation operator ($\odot$) is provided, w(indices) = w(indices) $\odot$ u. More explicitly:

$$\begin{aligned} \mathsf{w}(\mathsf{indices}[i]) &= \mathsf{u}(i),\ \forall\ i\ :\ 0 \leq i < \mathsf{nindices}, \quad \text{or} \\ \mathsf{w}(\mathsf{indices}[i]) &= \mathsf{w}(\mathsf{indices}[i]) \odot \mathsf{u}(i),\ \forall\ i\ :\ 0 \leq i < \mathsf{nindices}. \end{aligned}$$

Logically, this operation occurs in three steps:

Setup The internal vectors and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to three argument vectors are used in the GrB_assign operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)
3. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i, u_i)\} \rangle$

The argument vectors and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}(\mathsf{u})$.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_assign ends and the domain mismatch error listed above is returned.

From the arguments, the internal vectors, mask and index array used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.
2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:
   (a) If $\mathsf{mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i, \ \forall \ i : 0 \leq i < \mathbf{size}(\mathsf{w})\} \rangle$.
   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\} \rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.
4. The internal index array, $\widetilde{\boldsymbol{I}}$, is computed from argument indices as follows:
   (a) If $\mathsf{indices} = \mathsf{GrB\_ALL}$, then $\widetilde{\boldsymbol{I}}[i] = i, \ \forall \ i : 0 \leq i < \mathsf{nindices}$.
   (b) Otherwise, $\widetilde{\boldsymbol{I}}[i] = \mathsf{indices}[i], \ \forall \ i : 0 \leq i < \mathsf{nindices}$.

The internal vector and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}})$

2. nindices $= \mathbf{size}(\widetilde{\mathbf{u}})$.

If any compatibility rule above is violated, execution of GrB_assign ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the assign and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the elements from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{w}}$.
- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$\widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(\widetilde{\boldsymbol{I}}[i], \widetilde{\mathbf{u}}(i)) \forall i, 0 \leq i < \mathsf{nindices} : i \in \mathbf{ind}(\widetilde{\mathbf{u}})\}\rangle.$$

At this point, if any value of $\widetilde{\boldsymbol{I}}[i]$ is outside the valid range of indices for vector $\widetilde{\mathbf{w}}$, computation ends and the method returns the index-out-of-bounds error listed above. In GrB_NONBLOCKING mode, the error can be deferred until a sequence-terminating GrB_wait() is called. Regardless, the result vector, w, is invalid from this point forward in the sequence.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}}$ is defined as

  $$\widetilde{\mathbf{z}} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{w}}))) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The above expression defines the structure of vector $\widetilde{\mathbf{z}}$ as follows: We start with the structure of $\widetilde{\mathbf{w}}$ ($\mathbf{ind}(\widetilde{\mathbf{w}})$) and remove from it all the indices of $\widetilde{\mathbf{w}}$ that are in the set of indices being assigned ($\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{w}})$). Finally, we add the structure of $\widetilde{\mathbf{t}}$ ($\mathbf{ind}(\widetilde{\mathbf{t}})$).

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

  $$z_i = \widetilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  $$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\widetilde{\mathbf{t}}),$$

  where the difference operator refers to set difference.

- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

  $$\langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \ \forall \ i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

  $$z_i = \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})),$$

  $$z_i = \widetilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  $$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\widetilde{\mathbf{z}}$ are written into the final result vector w, using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w, is defined as,

$$\mathbf{L}(\mathsf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, w, and elements of w that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathsf{w}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.7.2 assign: Standard matrix variant

Assign values (and implied zeros) from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices. The dimensions of the input matrix are the same size as the row and column index arrays provided.

C Syntax

```
GrB_Info GrB_assign(GrB_Matrix            C,
                    const GrB_Matrix      Mask,
                    const GrB_BinaryOp    accum,
                    const GrB_Matrix      A,
                    const GrB_Index      *row_indices,
                    GrB_Index             nrows,
                    const GrB_Index      *col_indices,
                    GrB_Index             ncols,
                    const GrB_Descriptor  desc);
```

Parameters

C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the assign operation. On output, the matrix holds the results of the operation.

Mask (IN) An optional "write" mask that controls which results from this operation are stored into the output matrix C. The mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default matrix is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

A (IN) The GraphBLAS matrix whose contents are assigned to a subset of C.

row_indices (IN) Pointer to the ordered set (array) of indices corresponding to the rows of C that are assigned. If all rows of C are to be assigned in order from 0 to nrows $-$ 1, then GrB_ALL can be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation. If this array contains duplicate values, it implies assignment of more than one value to the same location which leads to undefined results.

nrows (IN) The number of values in the row_indices array. Must be equal to **nrows**(A) if A is not tranposed, or equal to **ncols**(A) if A is transposed.

col_indices (IN) Pointer to the ordered set (array) of indices corresponding to the columns of C that are assigned. If all columns of C are to be assigned in order from 0 to ncols $-$ 1, then GrB_ALL should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation. If this array contains duplicate values, it implies assignment of more than one value to the same location which leads to undefined results.

ncols (IN) The number of values in col_indices array. Must be equal to **ncols**(A) if A is not tranposed, or equal to **nrows**(A) if A is transposed.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_SCMP	Use the structural complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

Return Values

- GrB_SUCCESS: In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.
- GrB_PANIC: Unknown internal error.
- GrB_INVALID_OBJECT: This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.
- GrB_OUT_OF_MEMORY: Not enough memory available for the operation.
- GrB_UNINITIALIZED_OBJECT: One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).
- GrB_INDEX_OUT_OF_BOUNDS: A value in row_indices is greater than or equal to **nrows**(C), or a value in col_indices is greater than or equal to **ncols**(C). In non-blocking mode, this can be reported as an execution error.
- GrB_DIMENSION_MISMATCH: Mask and C dimensions are incompatible, nrows $\neq$ **nrows**(A), or ncols $\neq$ **ncols**(A).
- GrB_DOMAIN_MISMATCH: The domains of the various matrices are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.
- GrB_NULL_POINTER: Either argument row_indices is a NULL pointer, argument col_indices is a NULL pointer, or both.

Description

This variant of GrB_assign computes the result of assigning the contents of A to a subset of rows and columns in C in a specified order: $\mathsf{C}(\mathsf{row\_indices}, \mathsf{col\_indices}) = \mathsf{A}$; or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C}(\mathsf{row\_indices}, \mathsf{col\_indices}) = \mathsf{C}(\mathsf{row\_indices}, \mathsf{col\_indices}) \odot \mathsf{A}$. More explicitly (not accounting for an optional transpose of A):

$$
\begin{aligned}
\mathsf{C}(\mathsf{row\_indices}[i], \mathsf{col\_indices}[j]) &= \mathsf{A}(i,j),\ \forall\ i,j\ :\ 0 \leq i < \mathsf{nrows},\ 0 \leq j < \mathsf{ncols}, \text{ or} \\
\mathsf{C}(\mathsf{row\_indices}[i], \mathsf{col\_indices}[j]) &= \mathsf{C}(\mathsf{row\_indices}[i], \mathsf{col\_indices}[j]) \odot \mathsf{A}(i,j), \\
&\qquad \forall\ (i,j)\ :\ 0 \leq i < \mathsf{nrows},\ 0 \leq j < \mathsf{ncols}
\end{aligned}
$$

Logically, this operation occurs in three steps:

- Setup: The internal matrices and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to three argument matrices are used in the GrB_assign operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij})\}\rangle$
2. $\mathsf{Mask} = \langle \mathbf{D}(\mathsf{Mask}), \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \mathbf{L}(\mathsf{Mask}) = \{(i, j, M_{ij})\}\rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\}\rangle$

The argument matrices and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}(\mathsf{A})$.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_assign ends and the domain mismatch error listed above is returned.

From the arguments, the internal matrices, mask, and index arrays used in the computation are formed ($\leftarrow$ denotes copy):

1. Matrix $\widetilde{\mathbf{C}} \leftarrow \mathsf{C}$.
2. Two-dimensional mask $\widetilde{\mathbf{M}}$ is computed from argument Mask as follows:
   (a) If $\mathsf{Mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathsf{C}), 0 \leq j < \mathbf{ncols}(\mathsf{C})\}\rangle$.
   (b) Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathsf{Mask}) \wedge (\mathsf{bool})\mathsf{Mask}(i, j) = \mathsf{true}\}\rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}$.
3. Matrix $\widetilde{\mathbf{A}} \leftarrow \mathsf{desc[GrB\_INP0].GrB\_TRAN}\ ?\ \mathsf{A}^T : \mathsf{A}$.
4. The internal row index array, $\widetilde{\boldsymbol{I}}$, is computed from argument row_indices as follows:
   (a) If $\mathsf{row\_indices} = \mathsf{GrB\_ALL}$, then $\widetilde{\boldsymbol{I}}[i] = i, \forall i : 0 \leq i < \mathsf{nrows}$.
   (b) Otherwise, $\widetilde{\boldsymbol{I}}[i] = \mathsf{row\_indices}[i], \forall i : 0 \leq i < \mathsf{nrows}$.

5. The internal column index array, $\widetilde{\boldsymbol{J}}$, is computed from argument col_indices as follows:

   (a) If col_indices = GrB_ALL, then $\widetilde{\boldsymbol{J}}[j] = j, \forall j : 0 \leq j <$ ncols.

   (b) Otherwise, $\widetilde{\boldsymbol{J}}[j] =$ col_indices$[j], \ \forall\ j : 0 \leq j <$ ncols.

The internal matrices and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.

2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.

3. $\mathbf{nrows}(\widetilde{\mathbf{A}}) = \mathsf{nrows}$.

4. $\mathbf{ncols}(\widetilde{\mathbf{A}}) = \mathsf{ncols}$.

If any compatibility rule above is violated, execution of GrB_assign ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the assign and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{T}}$: The matrix holding the contents from $\widetilde{\mathbf{A}}$ in their destination locations relative to $\widetilde{\mathbf{C}}$.

- $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

The intermediate matrix, $\widetilde{\mathbf{T}}$, is created as follows:

$$\begin{aligned}\widetilde{\mathbf{T}} = \langle &\mathbf{D}(\mathsf{A}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}),\\ &\{(\widetilde{\boldsymbol{I}}[i], \widetilde{\boldsymbol{J}}[j], \widetilde{\mathbf{A}}(i,j))\ \forall\ (i,j),\ 0 \leq i < \mathsf{nrows},\ 0 \leq j < \mathsf{ncols} : (i,j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\}\rangle.\end{aligned}$$

At this point, if any value in the $\widetilde{\boldsymbol{I}}$ array is not in the range $[0,\ \mathbf{nrows}(\widetilde{\mathbf{C}}))$ or any value in the $\widetilde{\boldsymbol{J}}$ array is not in the range $[0,\ \mathbf{ncols}(\widetilde{\mathbf{C}}))$, the execution of GrB_assign ends and the index out-of-bounds error listed above is generated. In GrB_NONBLOCKING mode, the error can be deferred until a sequence-terminating GrB_wait() is called. Regardless, the result matrix C is invalid from this point forward in the sequence.

The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows:

- If accum = GrB_NULL, then $\widetilde{\mathbf{Z}}$ is defined as

$$\begin{aligned}\widetilde{\mathbf{Z}} \quad = \quad &\langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}),\\ &\{(i,j,Z_{ij}) \forall (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\{(\widetilde{\boldsymbol{I}}[k], \widetilde{\boldsymbol{J}}[l]), \forall k,l\} \cap \mathbf{ind}(\widetilde{\mathbf{C}}))) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\}\rangle.\end{aligned}$$

The above expression defines the structure of matrix $\widetilde{\mathbf{Z}}$ as follows: We start with the structure of $\widetilde{\mathbf{C}}$ ($\mathbf{ind}(\widetilde{\mathbf{C}})$) and remove from it all the indices of $\widetilde{\mathbf{C}}$ that are in the set of indices being assigned ($\{(\widetilde{\boldsymbol{I}}[k], \widetilde{\boldsymbol{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\widetilde{\mathbf{C}})$). Finally, we add the structure of $\widetilde{\mathbf{T}}$ ($\mathbf{ind}(\widetilde{\mathbf{T}})$).

The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\{(\widetilde{\boldsymbol{I}}[k], \widetilde{\boldsymbol{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

$$Z_{ij} = \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in \mathbf{ind}(\widetilde{\mathbf{T}}),$$

where the difference operator refers to set difference.

- If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$\langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i,j,Z_{ij}) \forall (i,j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j) \odot \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}})),$$

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

$$Z_{ij} = \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up matrix $\widetilde{\mathbf{Z}}$ are written into the final result matrix C, using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in C on input to this operation are deleted and the content of the new output matrix, C, is defined as,

$$\mathbf{L}(\mathsf{C}) = \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{Z}}$ indicated by the mask are copied into the result matrix, C, and elements of C that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{C}) = \{(i,j,C_{ij}) : (i,j) \in (\mathbf{ind}(\mathsf{C}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{M}}))\} \cup \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.7.3 assign: Column variant

Assign the contents a vector to a subset of elements in one column of a matrix. Note that since the output cannot be transposed, a different variant of assign is provided to assign to a row of a matrix.

C Syntax

```
GrB_Info GrB_assign(GrB_Matrix            C,
                    const GrB_Vector      mask,
                    const GrB_BinaryOp    accum,
                    const GrB_Vector      u,
                    const GrB_Index      *row_indices,
                    GrB_Index             nrows,
                    GrB_Index             col_index,
                    const GrB_Descriptor  desc);
```

Parameters

C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the assign operation. On output, this matrix holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the specified column of the output matrix C. The mask dimensions must match those of a single column of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

u (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column of C.

row_indices (IN) Pointer to the ordered set (array) of indices corresponding to the locations in the specified column of C that are to be assigned. If all elements of the column in C are to be assigned in order from index 0 to $\text{nrows} - 1$, then GrB_ALL should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation. If this array contains duplicate values, it implies in assignment of more than one value to the same location which leads to undefined results.

nrows (IN) The number of values in row_indices array. Must be equal to **size**(u).

col_index (IN) The index of the column in C to assign. Must be in the range $[0, \mathbf{ncols}(\mathsf{C}))$.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output column in C is cleared (all elements removed) before result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector or matrix parameters).

GrB_INVALID_INDEX col_index is outside the allowable range (i.e., greater than **ncols**(C)).

GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to **nrows**(C). In non-blocking mode, this can be reported as an execution error.

GrB_DIMENSION_MISMATCH mask size and number of rows in C are not the same, or nrows $\neq$ **size**(u).

GrB_DOMAIN_MISMATCH The domains of the matrix and vector are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.

GrB_NULL_POINTER Argument row_indices is a NULL pointer.

Description

This variant of GrB_assign computes the result of assigning a subset of locations in a column of a GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector: $\mathsf{C}(:,\mathsf{col\_index}) = \mathsf{u}$; or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C}(:,\mathsf{col\_index}) = \mathsf{C}(:,\mathsf{col\_index}) \odot \mathsf{u}$. Taking order of row_indices into account, it is more explicitly written as:

$$\begin{aligned}
&\mathsf{C}(\mathsf{row_indices}[i],\mathsf{col_index}) = \mathsf{u}(i),\ \forall\ i\ :\ 0 \leq i < \mathsf{nrows},\ \text{or}\\
&\mathsf{C}(\mathsf{row_indices}[i],\mathsf{col_index}) = \mathsf{C}(\mathsf{row_indices}[i],\mathsf{col_index}) \odot \mathsf{u}(i),\ \forall\ i\ :\ 0 \leq i < \mathsf{nrows}.
\end{aligned}$$

Logically, this operation occurs in three steps:

Setup The internal matrices, vectors and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to three argument vectors and matrices are used in this GrB_assign operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i,j,C_{ij})\}\rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i,m_i)\}\rangle$ (optional)
3. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i,u_i)\}\rangle$

The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}(\mathsf{u})$.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_assign ends and the domain mismatch error listed above is returned.

The col_index parameter is checked for a valid value. The following condition must hold:

1. $0 \leq \mathsf{col\_index} < \mathbf{ncols}(\mathsf{C})$

If the rule above is violated, execution of GrB_assign ends and the invalid index error listed above is returned.

From the arguments, the internal vectors, mask, and index array used in the computation are formed ($\leftarrow$ denotes copy):

1. The vector, $\widetilde{\mathbf{c}}$, is extracted from a column of C as follows:

$$\widetilde{\mathbf{c}} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \{(i, C_{ij})\ \forall\ i : 0 \leq i < \mathbf{nrows}(\mathsf{C}), j = \mathsf{col\_index}, (i,j) \in \mathbf{ind}(\mathsf{C})\}\rangle$$

2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:
   (a) If $\mathsf{mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathsf{C}), \{i,\ \forall\ i : 0 \leq i < \mathbf{nrows}(\mathsf{C})\}\rangle$.
   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\}\rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg\widetilde{\mathbf{m}}$.
3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.
4. The internal row index array, $\widetilde{\boldsymbol{I}}$, is computed from argument row_indices as follows:
   (a) If $\mathsf{row\_indices} = \mathsf{GrB\_ALL}$, then $\widetilde{\boldsymbol{I}}[i] = i,\ \forall\ i : 0 \leq i < \mathsf{nrows}$.
   (b) Otherwise, $\widetilde{\boldsymbol{I}}[i] = \mathsf{row\_indices}[i],\ \forall\ i : 0 \leq i < \mathsf{nrows}$.

The internal vectors, matrices, and masks are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{c}}) = \mathbf{size}(\widetilde{\mathbf{m}})$
2. $\mathsf{nrows} = \mathbf{size}(\widetilde{\mathbf{u}})$.

If any compatibility rule above is violated, execution of GrB_assign ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the assign and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the elements from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{c}}$.
- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$\widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\widetilde{\mathbf{c}}), \{(\widetilde{\boldsymbol{I}}[i], \widetilde{\mathbf{u}}(i))\ \forall\ i,\ 0 \leq i < \mathsf{nrows} : i \in \mathbf{ind}(\widetilde{\mathbf{u}})\}\rangle.$$

At this point, if any value of $\widetilde{\boldsymbol{I}}[i]$ is outside the valid range of indices for vector $\widetilde{\mathbf{c}}$, computation ends and the method returns the index out-of-bounds error listed above. In GrB_NONBLOCKING mode, the error can be deferred until a sequence-terminating GrB_wait() is called. Regardless, the result matrix, C, is invalid from this point forward in the sequence.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}}$ is defined as

$$\widetilde{\mathbf{z}} = \langle \mathbf{D}(\mathsf{C}), \mathbf{size}(\widetilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\widetilde{\mathbf{c}}) - (\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{c}}))) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The above expression defines the structure of vector $\widetilde{\mathbf{z}}$ as follows: We start with the structure of $\widetilde{\mathbf{c}}$ ($\mathbf{ind}(\widetilde{\mathbf{c}})$) and remove from it all the indices of $\widetilde{\mathbf{c}}$ that are in the set of indices being assigned ($\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{c}})$). Finally, we add the structure of $\widetilde{\mathbf{t}}$ ($\mathbf{ind}(\widetilde{\mathbf{t}})$).

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{c}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{c}}) - (\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{c}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\widetilde{\mathbf{t}}),$$

  where the difference operator refers to set difference.

- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$\langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{c}}), \{(i, z_i) \ \forall \ i \in \mathbf{ind}(\widetilde{\mathbf{c}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{c}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{c}})),$$

$$z_i = \widetilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{c}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{c}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{c}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up the $\widetilde{\mathbf{z}}$ vector are written into the column of the final result matrix, C(:, col_index). This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in C(:, col_index) on input to this operation are deleted and the new contents of the column is given by:

$$\mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij}) : j \neq \mathsf{col\_index}\} \cup \{(i, \mathsf{col\_index}, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the column of the final result matrix, C(:, col_index), and elements of this column that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned}
\mathbf{L}(\mathsf{C}) \;\; = \;\; & \{(i, j, C_{ij}) : j \neq \mathsf{col_index}\} \cup \\
& \{(i, \mathsf{col_index}, \widetilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\widetilde{\mathbf{c}}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \\
& \{(i, \mathsf{col_index}, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.
\end{aligned}$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

4.3.7.4 assign: Row variant

Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the output cannot be transposed, a different variant of assign is provided to assign to a column of a matrix.

C Syntax

```
GrB_Info GrB_assign(GrB_Matrix             C,
                    const GrB_Vector       mask,
                    const GrB_BinaryOp     accum,
                    const GrB_Vector       u,
                    GrB_Index              row_index,
                    const GrB_Index       *col_indices,
                    GrB_Index              ncols,
                    const GrB_Descriptor   desc);
```

Parameters

C (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values that may be accumulated with the result of the assign operation. On output, this matrix holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the specified row of the output matrix C. The mask dimensions must match those of a single row of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

u (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of C.

row_index (IN) The index of the row in C to assign. Must be in the range $[0, \mathbf{nrows}(\mathsf{C}))$.

col_indices (IN) Pointer to the ordered set (array) of indices corresponding to the locations in the specified row of C that are to be assigned. If all elements of the row in C are to be assigned in order from index 0 to ncols $-$ 1, then GrB_ALL should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation. If this array contains duplicate values, it implies in assignment of more than one value to the same location which leads to undefined results.

ncols (IN) The number of values in col_indices array. Must be equal to **size**(u).

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output row in C is cleared (all elements removed) before result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector or matrix parameters).

GrB_INVALID_INDEX row_index is outside the allowable range (i.e., greater than **nrows**(C)).

GrB_INDEX_OUT_OF_BOUNDS A value in col_indices is greater than or equal to **ncols**(C). In non-blocking mode, this can be reported as an execution error.

GrB_DIMENSION_MISMATCH mask size and number of columns in C are not the same, or ncols $\neq$ **size**(u).

GrB_DOMAIN_MISMATCH The domains of the matrix and vector are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.

GrB_NULL_POINTER Argument col_indices is a NULL pointer.

Description

This variant of GrB_assign computes the result of assigning a subset of locations in a row of a GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector: $\mathsf{C}(\mathsf{row\_index},:) = \mathsf{u}$; or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C}(\mathsf{row\_index},:) = \mathsf{C}(\mathsf{row\_index},:) \odot \mathsf{u}$. Taking order of col_indices into account it is more explicitly written as:

$$\mathsf{C}(\mathsf{row\_index}, \mathsf{col\_indices}[j]) = \mathsf{u}(j),\ \forall\ j\ :\ 0 \leq j < \mathsf{ncols},\ \text{or}$$
$$\mathsf{C}(\mathsf{row\_index}, \mathsf{col\_indices}[j]) = \mathsf{C}(\mathsf{row\_index}, \mathsf{col\_indices}[j]) \odot \mathsf{u}(j),\ \forall\ j\ :\ 0 \leq j < \mathsf{ncols}$$

Logically, this operation occurs in three steps:

Setup The internal matrices, vectors and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to three argument vectors and matrices are used in this GrB_assign operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij})\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)
3. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i, u_i)\} \rangle$

The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}(\mathsf{u})$.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_assign ends and the domain mismatch error listed above is returned.

The row_index parameter is checked for a valid value. The following condition must hold:

1. $0 \;\leq\; \mathsf{row\_index} \;<\; \mathbf{nrows}(\mathsf{C})$

If the rule above is violated, execution of GrB_assign ends and the invalid index error listed above is returned.

From the arguments, the internal vectors, mask, and index array used in the computation are formed ($\leftarrow$ denotes copy):

1. The vector, $\widetilde{\mathbf{c}}$, is extracted from a row of C as follows:

$$\widetilde{\mathbf{c}} = \langle \mathbf{D}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \{(j, C_{ij}) \; \forall \; j : 0 \leq j < \mathbf{ncols}(\mathsf{C}), i = \mathsf{row\_index}, (i,j) \in \mathbf{ind}(\mathsf{C})\}\rangle$$

2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:
   (a) If $\mathsf{mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathsf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathsf{C})\}\rangle$.
   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\}\rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg\widetilde{\mathbf{m}}$.
3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.
4. The internal column index array, $\widetilde{\boldsymbol{J}}$, is computed from argument col_indices as follows:
   (a) If $\mathsf{col\_indices} = \mathsf{GrB\_ALL}$, then $\widetilde{\boldsymbol{J}}[j] = j, \; \forall \; j : 0 \leq j < \mathsf{ncols}$.
   (b) Otherwise, $\widetilde{\boldsymbol{J}}[j] = \mathsf{col\_indices}[j], \; \forall \; j : 0 \leq j < \mathsf{ncols}$.

The internal vectors, matrices, and masks are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{c}}) = \mathbf{size}(\widetilde{\mathbf{m}})$
2. $\mathsf{ncols} = \mathbf{size}(\widetilde{\mathbf{u}})$.

If any compatibility rule above is violated, execution of GrB_assign ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the assign and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the elements from $\widetilde{\mathbf{u}}$ in their destination locations relative to $\widetilde{\mathbf{c}}$.
- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$\widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\widetilde{\mathbf{c}}), \{(\widetilde{\boldsymbol{J}}[j], \widetilde{\mathbf{u}}(j)) \; \forall \; j, \;\; 0 \leq j < \mathsf{ncols} : j \in \mathbf{ind}(\widetilde{\mathbf{u}})\}\rangle.$$

At this point, if any value of $\widetilde{\boldsymbol{J}}[j]$ is outside the valid range of indices for vector $\widetilde{\mathbf{c}}$, computation ends and the method returns the index out-of-bounds error listed above. In GrB_NONBLOCKING mode, the error can be deferred until a sequence-terminating GrB_wait() is called. Regardless, the result matrix, C, is invalid from this point forward in the sequence.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}}$ is defined as

$$\widetilde{\mathbf{z}} = \langle \mathbf{D}(\mathsf{C}), \mathbf{size}(\widetilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\widetilde{\mathbf{c}}) - (\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{c}}))) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The above expression defines the structure of vector $\widetilde{\mathbf{z}}$ as follows: We start with the structure of $\widetilde{\mathbf{c}}$ ($\mathbf{ind}(\widetilde{\mathbf{c}})$) and remove from it all the indices of $\widetilde{\mathbf{c}}$ that are in the set of indices being assigned ($\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{c}})$). Finally, we add the structure of $\widetilde{\mathbf{t}}$ ($\mathbf{ind}(\widetilde{\mathbf{t}})$).

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{c}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{c}}) - (\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{c}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\widetilde{\mathbf{t}}),$$

  where the difference operator refers to set difference.

- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$\langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{c}}), \{(j, z_j) \ \forall \ j \in \mathbf{ind}(\widetilde{\mathbf{c}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

$$z_j = \widetilde{\mathbf{c}}(j) \odot \widetilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{c}})),$$

$$z_j = \widetilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\widetilde{\mathbf{c}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{c}}))),$$

$$z_j = \widetilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{c}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up the $\widetilde{\mathbf{z}}$ vector are written into the column of the final result matrix, $\mathsf{C}(\mathsf{row\_index}, :)$. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in $\mathsf{C}(\mathsf{row\_index}, :)$ on input to this operation are deleted and the new contents of the column is given by:

$$\mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij}) : i \neq \mathsf{row\_index}\} \cup \{(\mathsf{row\_index}, j, z_j) : j \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the column of the final result matrix, $\mathsf{C}(\mathsf{row\_index}, :)$, and elements of this column that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} \mathbf{L}(\mathsf{C}) \quad = \quad & \{(i, j, C_{ij}) : i \neq \mathsf{row\_index}\} \cup \\ & \{(\mathsf{row\_index}, j, \widetilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\widetilde{\mathbf{c}}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \\ & \{(\mathsf{row\_index}, j, z_j) : j \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}. \end{aligned}$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

4.3.7.5 assign: Constant vector variant

Assign the same value to a specified subset of vector elements. With the use of GrB_ALL, the entire destination vector can be filled with the constant.

C Syntax

```
GrB_Info GrB_assign(GrB_Vector             w,
                    const GrB_Vector       mask,
                    const GrB_BinaryOp     accum,
                    <type>                 val,
                    const GrB_Index       *indices,
                    GrB_Index              nindices,
                    const GrB_Descriptor   desc);
```

Parameters

- **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the assign operation. On output, this vector holds the results of the operation.

- **mask** (IN) An optional "write" mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w and the domain of the mask vector must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

- **accum** (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

- **val** (IN) Scalar value to assign to (a subset of) w.

- **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in w that are to be assigned. If all elements of w are to be assigned in order from 0 to nindices $-$ 1, then GrB_ALL should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation. In this variant, the specific order of the values in the array has no effect on the result. Unlike other variants, if there are duplicated values in this array the result is still defined.

nindices (IN) The number of values in indices array. Must be in the range: $[0, \mathbf{size}(w)]$. If nindices is zero, the operation becomes a NO-OP.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector parameters).

GrB_INDEX_OUT_OF_BOUNDS A value in indices is greater than or equal to **size**(w). In non-blocking mode, this can be reported as an execution error.

GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or nindices is not less than **size**(w).

GrB_DOMAIN_MISMATCH The domains of the vector and scalar are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.

GrB_NULL_POINTER Argument indices is a NULL pointer.

Description

This variant of GrB_assign computes the result of assigning a constant scalar value to locations in a destination GraphBLAS vector: $\mathsf{w}(\mathsf{indices}) = \mathsf{val}$; or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{w}(\mathsf{indices}) = \mathsf{w}(\mathsf{indices}) \odot \mathsf{val}$. More explicitly:

$$\begin{aligned} \mathsf{w}(\mathsf{indices}[i]) &= \mathsf{val},\ \forall\ i : 0 \le i < \mathsf{nindices}, \quad \text{or} \\ \mathsf{w}(\mathsf{indices}[i]) &= \mathsf{w}(\mathsf{indices}[i]) \odot \mathsf{val},\ \forall\ i : 0 \le i < \mathsf{nindices}. \end{aligned}$$

Logically, this operation occurs in three steps:

Setup The internal vectors and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to two argument vectors are used in the GrB_assign operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)

The argument scalar, vectors, and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}(\mathsf{val})$.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{val})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_assign ends and the domain mismatch error listed above is returned.

From the arguments, the internal vectors, mask and index array used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.
2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:

(a) If mask = GrB_NULL, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i,\ \forall\ i : 0 \le i < \mathbf{size}(\mathsf{w})\} \rangle$.

(b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\} \rangle$.

(c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.

3. The internal index array, $\widetilde{\boldsymbol{I}}$, is computed from argument indices as follows:

(a) If indices = GrB_ALL, then $\widetilde{\boldsymbol{I}}[i] = i,\ \forall\ i : 0 \le i < \mathsf{nindices}$.

(b) Otherwise, $\widetilde{\boldsymbol{I}}[i] = \mathsf{indices}[i],\ \forall\ i : 0 \le i < \mathsf{nindices}$.

The internal vector and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}})$
2. $0 \le \mathsf{nindices} \le \mathbf{size}(\widetilde{\mathbf{w}})$.

If any compatibility rule above is violated, execution of GrB_assign ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the assign and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the copies of the scalar val in their destination locations relative to $\widetilde{\mathbf{w}}$.
- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$\widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathsf{val}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(\widetilde{\boldsymbol{I}}[i], \mathsf{val})\ \forall\ i,\ 0 \le i < \mathsf{nindices}\} \rangle.$$

If $\widetilde{\boldsymbol{I}}$ is empty, this operation results in an empty vector, $\widetilde{\mathbf{t}}$. Otherwise, if any value in the $\widetilde{\boldsymbol{I}}$ array is not in the range $[0,\ \mathbf{size}(\widetilde{\mathbf{w}}))$, the execution of GrB_assign ends and the index out-of-bounds error listed above is generated. In GrB_NONBLOCKING mode, the error can be deferred until a sequence-terminating GrB_wait() is called. Regardless, the result vector, w, is invalid from this point forward in the sequence.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}}$ is defined as

$$\widetilde{\mathbf{z}} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{w}}))) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

  The above expression defines the structure of vector $\widetilde{\mathbf{z}}$ as follows: We start with the structure of $\widetilde{\mathbf{w}}$ ($\mathbf{ind}(\widetilde{\mathbf{w}})$) and remove from it all the indices of $\widetilde{\mathbf{w}}$ that are in the set of indices being assigned ($\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{w}})$). Finally, we add the structure of $\widetilde{\mathbf{t}}$ ($\mathbf{ind}(\widetilde{\mathbf{t}})$).

The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\{\widetilde{\boldsymbol{I}}[k], \forall k\} \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\widetilde{\mathbf{t}}),$$

where the difference operator refers to set difference. We note that in this case of assigning a constant, $\{\widetilde{\boldsymbol{I}}[k], \forall k\}$ and $\mathbf{ind}(\widetilde{\mathbf{t}})$ are identical.

- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$\langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \ \forall \ i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})),$$

$$z_i = \widetilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\widetilde{\mathbf{z}}$ are written into the final result vector w, using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w, is defined as,

$$\mathbf{L}(\mathsf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, w, and elements of w that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathsf{w}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.7.6 assign: Constant matrix variant

Assign the same value to a specified subset of matrix elements. With the use of GrB_ALL, the entire destination matrix can be filled with the constant.

C Syntax

```
        GrB_Info GrB_assign(GrB_Matrix            C,
                            const GrB_Matrix      Mask,
                            const GrB_BinaryOp    accum,
                            <type>                val,
                            const GrB_Index      *row_indices,
                            GrB_Index             nrows,
                            const GrB_Index      *col_indices,
                            GrB_Index             ncols,
                            const GrB_Descriptor  desc);
```

Parameters

C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the assign operation. On output, the matrix holds the results of the operation.

Mask (IN) An optional "write" mask that controls which results from this operation are stored into the output matrix C. The mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default matrix is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

val (IN) Scalar value to assign to (a subset of) C.

row_indices (IN) Pointer to the ordered set (array) of indices corresponding to the rows of C that are assigned. If all rows of C are to be assigned in order from 0 to $\text{nrows} - 1$, then GrB_ALL can be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation. Unlike other variants, if there are duplicated values in this array the result is still defined.

nrows (IN) The number of values in row_indices array. Must be in the range: $[0, \mathbf{nrows}(\mathsf{C})]$. If nrows is zero, the operation becomes a NO-OP.

col_indices (IN) Pointer to the ordered set (array) of indices corresponding to the columns of C that are assigned. If all columns of C are to be assigned in order from 0 to $\text{ncols}-1$, then GrB_ALL should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation. Unlike other variants, if there are duplicated values in this array the result is still defined.

ncols (IN) The number of values in col_indices array. Must be in the range: $[0, \mathbf{ncols}(\mathsf{C})]$. If ncols is zero, the operation becomes a NO-OP.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_SCMP	Use the structural complement of Mask.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector parameters).

GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to $\mathbf{nrows}(\mathsf{C})$, or a value in col_indices is greater than or equal to $\mathbf{ncols}(\mathsf{C})$. In non-blocking mode, this can be reported as an execution error.

GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrows is not less than $\mathbf{nrows}(\mathsf{C})$, or ncols is not less than $\mathbf{ncols}(\mathsf{C})$.

GrB_DOMAIN_MISMATCH The domains of the matrix and scalar are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.

GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices is a NULL pointer, or both.

Description

This variant of GrB_assign computes the result of assigning a constant scalar value to locations in a destination GraphBLAS matrix: $\mathsf{C}(\mathsf{row\_indices}, \mathsf{col\_indices}) = \mathsf{val}$; or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C}(\mathsf{row\_indices}, \mathsf{col\_indices}) = \mathsf{w}(\mathsf{row\_indices}, \mathsf{col\_indices}) \odot \mathsf{val}$. More explicitly:

$$
\begin{aligned}
\mathsf{C}(\mathsf{row\_indices}[i], \mathsf{col\_indices}[j]) &= \mathsf{val}, \text{ or} \\
\mathsf{C}(\mathsf{row\_indices}[i], \mathsf{col\_indices}[j]) &= \mathsf{C}(\mathsf{row\_indices}[i], \mathsf{col\_indices}[j]) \odot \mathsf{val} \\
&\qquad\qquad\qquad \forall\ (i,j)\ :\ 0 \leq i < \mathsf{nrows},\ 0 \leq j < \mathsf{ncols}
\end{aligned}
$$

Logically, this operation occurs in three steps:

Setup The internal vectors and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to two argument matrices are used in the GrB_assign operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i,j,C_{ij})\} \rangle$
2. $\mathsf{Mask} = \langle \mathbf{D}(\mathsf{Mask}), \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \mathbf{L}(\mathsf{Mask}) = \{(i,j,M_{ij})\} \rangle$ (optional)

The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}(\mathsf{val})$.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{val})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_assign ends and the domain mismatch error listed above is returned.

From the arguments, the internal matrices, index arrays, and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Matrix $\widetilde{\mathbf{C}} \leftarrow \mathsf{C}$.

2. Two-dimensional mask $\widetilde{\mathbf{M}}$ is computed from argument Mask as follows:

   (a) If $\mathsf{Mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \{(i,j), \forall i,j : 0 \leq i < \mathbf{nrows}(\mathsf{C}), 0 \leq j < \mathbf{ncols}(\mathsf{C})\}\rangle$.

   (b) Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \{(i,j) : (i,j) \in \mathbf{ind}(\mathsf{Mask}) \wedge (\mathsf{bool})\mathsf{Mask}(i,j) = \mathsf{true}\}\rangle$.

   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}$.

3. The internal row index array, $\widetilde{\boldsymbol{I}}$, is computed from argument row_indices as follows:

   (a) If $\mathsf{row\_indices} = \mathsf{GrB\_ALL}$, then $\widetilde{\boldsymbol{I}}[i] = i, \forall i : 0 \leq i < \mathsf{nrows}$.

   (b) Otherwise, $\widetilde{\boldsymbol{I}}[i] = \mathsf{row\_indices}[i], \forall i : 0 \leq i < \mathsf{nrows}$.

4. The internal column index array, $\widetilde{\boldsymbol{J}}$, is computed from argument col_indices as follows:

   (a) If $\mathsf{col\_indices} = \mathsf{GrB\_ALL}$, then $\widetilde{\boldsymbol{J}}[j] = j, \forall j : 0 \leq j < \mathsf{ncols}$.

   (b) Otherwise, $\widetilde{\boldsymbol{J}}[j] = \mathsf{col\_indices}[j], \forall j : 0 \leq j < \mathsf{ncols}$.

The internal matrix and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.

2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.

3. $0 \leq \mathsf{nrows} \leq \mathbf{nrows}(\widetilde{\mathbf{C}})$.

4. $0 \leq \mathsf{ncols} \leq \mathbf{ncols}(\widetilde{\mathbf{C}})$.

If any compatibility rule above is violated, execution of GrB_assign ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the assign and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{T}}$: The matrix holding the copies of the scalar val in their destination locations relative to $\widetilde{\mathbf{C}}$.

- $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

The intermediate matrix, $\widetilde{\mathbf{T}}$, is created as follows:

$$\begin{aligned}\widetilde{\mathbf{T}} = \langle &\mathbf{D}(\mathsf{val}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}),\\ &\{(\widetilde{\boldsymbol{I}}[i], \widetilde{\boldsymbol{J}}[j], \mathsf{val}) \ \forall \ (i,j),\ 0 \leq i < \mathsf{nrows},\ 0 \leq j < \mathsf{ncols}\}\rangle.\end{aligned}$$

If either $\widetilde{\boldsymbol{I}}$ or $\widetilde{\boldsymbol{J}}$ is empty, this operation results in an empty matrix, $\widetilde{\mathbf{T}}$. Otherwise, if any value in the $\widetilde{\boldsymbol{I}}$ array is not in the range $[0,\ \mathbf{nrows}(\widetilde{\mathbf{C}}))$ or any value in the $\widetilde{\boldsymbol{J}}$ array is not in the range $[0,\ \mathbf{ncols}(\widetilde{\mathbf{C}}))$, the execution of GrB_assign ends and the index out-of-bounds error listed above is generated. In GrB_NONBLOCKING mode, the error can be deferred until a sequence-terminating GrB_wait() is called. Regardless, the result matrix C is invalid from this point forward in the sequence.

The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows:

- If accum = GrB_NULL, then $\widetilde{\mathbf{Z}}$ is defined as

$$\begin{aligned}\widetilde{\mathbf{Z}} \;=\; & \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \\ & \{(i,j,Z_{ij}) \forall (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\{(\widetilde{\boldsymbol{I}}[k], \widetilde{\boldsymbol{J}}[l]), \forall k,l\} \cap \mathbf{ind}(\widetilde{\mathbf{C}}))) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\}\rangle.\end{aligned}$$

  The above expression defines the structure of matrix $\widetilde{\mathbf{Z}}$ as follows: We start with the structure of $\widetilde{\mathbf{C}}$ ($\mathbf{ind}(\widetilde{\mathbf{C}})$) and remove from it all the indices of $\widetilde{\mathbf{C}}$ that are in the set of indices being assigned ($\{(\widetilde{\boldsymbol{I}}[k], \widetilde{\boldsymbol{J}}[l]), \forall k,l\} \cap \mathbf{ind}(\widetilde{\mathbf{C}})$). Finally, we add the structure of $\widetilde{\mathbf{T}}$ ($\mathbf{ind}(\widetilde{\mathbf{T}})$).

  The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j),\ \text{if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\{(\widetilde{\boldsymbol{I}}[k], \widetilde{\boldsymbol{J}}[l]), \forall k,l\} \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

$$Z_{ij} = \widetilde{\mathbf{T}}(i,j),\ \text{if } (i,j) \in \mathbf{ind}(\widetilde{\mathbf{T}}),$$

  where the difference operator refers to set difference. We note that, in this particular case of assigning a constant to a matrix, the sets $\{(\widetilde{\boldsymbol{I}}[k], \widetilde{\boldsymbol{J}}[l]), \forall k,l\}$ and $\mathbf{ind}(\widetilde{\mathbf{T}})$ are identical.

- If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$\langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i,j,Z_{ij}) \forall (i,j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j) \odot \widetilde{\mathbf{T}}(i,j),\ \text{if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}})),$$

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j),\ \text{if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

$$Z_{ij} = \widetilde{\mathbf{T}}(i,j),\ \text{if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up matrix $\widetilde{\mathbf{Z}}$ are written into the final result matrix C, using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in C on input to this operation are deleted and the content of the new output matrix, C, is defined as,

$$\mathbf{L}(\mathsf{C}) = \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{Z}}$ indicated by the mask are copied into the result matrix, C, and elements of C that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathsf{C}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.8 apply: Apply a unary function to the elements of an object

Computes the transformation of the values of the elements of a vector or a matrix using a unary function.

4.3.8.1 apply: Vector variant

Computes the transformation of the values of the elements of a vector using a unary function.

C Syntax

```
GrB_Info GrB_apply(GrB_Vector             w,
                   const GrB_Vector       mask,
                   const GrB_BinaryOp     accum,
                   const GrB_UnaryOp      op,
                   const GrB_Vector       u,
                   const GrB_Descriptor   desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the apply operation. On output, this vector holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w and the domain of the mask vector must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) A unary operator applied to each element of input vector u.

u (IN) The GraphBLAS vector to which the unary function is applied.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector parameters).

GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the corresponding domains of the accumulating operation, mask, or unary function.

Description

This variant of GrB_apply computes the result of applying a unary function to the elements of a GraphBLAS vector: $\mathsf{w} = f(\mathsf{u})$; or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{w} = \mathsf{w} \odot f(\mathsf{u})$.

Logically, this operation occurs in three steps:

Setup The internal vectors and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to three argument vectors are used in this GrB_apply operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)
3. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i, u_i)\} \rangle$

The argument vectors and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$ of the unary operator.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}_{out}(\mathsf{op})$ of the unary operator must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.
4. $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}_{in}(\mathsf{op})$.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_apply ends and the domain mismatch error listed above is returned.

From the argument vectors, the internal vectors and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.
2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:
   (a) If $\mathsf{mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathsf{w})\} \rangle$.
   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\} \rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.
3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.

The internal vectors and masks are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}})$
2. $\mathbf{size}(\widetilde{\mathbf{u}}) = \mathbf{size}(\widetilde{\mathbf{w}})$.

If any compatibility rule above is violated, execution of GrB_apply ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the apply and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the result from applying the unary operator to the input vector $\widetilde{\mathbf{u}}$.
- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$\widetilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{size}(\widetilde{\mathbf{u}}), \mathbf{L}(\widetilde{\mathbf{t}}) = \{(i, f(\widetilde{\mathbf{u}}(i))) \forall i \in \mathbf{ind}(\widetilde{\mathbf{u}})\}\rangle,$$

where $f = \mathbf{f}(\mathsf{op})$.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$\widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \ \forall \ i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})),$$

$$z_i = \widetilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\widetilde{\mathbf{z}}$ are written into the final result vector w, using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w, is defined as,

$$\mathbf{L}(\mathsf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, w, and elements of w that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathsf{w}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.8.2 apply: Matrix variant

Computes the transformation of the values of the elements of a matrix using a unary function.

C Syntax

```
GrB_Info GrB_apply(GrB_Matrix            C,
                   const GrB_Matrix      Mask,
                   const GrB_BinaryOp    accum,
                   const GrB_UnaryOp     op,
                   const GrB_Matrix      A,
                   const GrB_Descriptor  desc);
```

Parameters

C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the apply operation. On output, the matrix holds the results of the operation.

Mask (IN) An optional "write" mask that controls which results from this operation are stored into the output matrix C. The mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default matrix is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) A unary operator applied to each element of input matrix A.

A (IN) The GraphBLAS matrix to which the unary function is applied.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_SCMP	Use the structural complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to **nrows**(A), or a value in col_indices is greater than or equal to **ncols**(A). In non-blocking mode, this can be reported as an execution error.

GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrows $\neq$ **nrows**(C), or ncols $\neq$ **ncols**(C).

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.

Description

This variant of GrB_apply computes the result of applying a unary function to the elements of a GraphBLAS matrix: $\mathsf{C} = f(\mathsf{A})$; or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C} = \mathsf{C} \odot f(\mathsf{A})$.

Logically, this operation occurs in three steps:

Setup The internal matrices and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to three argument matrices are used in the GrB_apply operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij})\}\rangle$
2. $\mathsf{Mask} = \langle \mathbf{D}(\mathsf{Mask}), \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \mathbf{L}(\mathsf{Mask}) = \{(i, j, M_{ij})\}\rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\}\rangle$

The argument matrices and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$ of the unary operator.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}_{out}(\mathsf{op})$ of the unary operator must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.
4. $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}_{in}(\mathsf{op})$ of the unary operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_apply ends and the domain mismatch error listed above is returned.

From the argument matrices, the internal matrices, mask, and index arrays used in the computation are formed ($\leftarrow$ denotes copy):

1. Matrix $\widetilde{\mathbf{C}} \leftarrow \mathsf{C}$.
2. Two-dimensional mask, $\widetilde{\mathbf{M}}$, is computed from argument Mask as follows:
   (a) If $\mathsf{Mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathsf{C}), 0 \leq j < \mathbf{ncols}(\mathsf{C})\}\rangle$.
   (b) Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \{(i, j) : (\mathsf{bool})\mathsf{Mask}(i, j) = \mathsf{true}\}\rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}$.

3. Matrix $\widetilde{\mathbf{A}} \leftarrow \mathsf{desc[GrB\_INP0].GrB\_TRAN}\ ?\ \mathsf{A}^T : \mathsf{A}$.

The internal matrices and mask are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.
4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.

If any compatibility rule above is violated, execution of GrB_apply ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the apply and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\widetilde{\mathbf{T}}$: The matrix holding the result from applying the unary operator to the input matrix $\widetilde{\mathbf{A}}$.
- $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

The intermediate matrix, $\widetilde{\mathbf{T}}$, is created as follows:

$$\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \mathbf{L}(\widetilde{\mathbf{T}}) = \{(i,j,f(\widetilde{\mathbf{A}}(i,j)))\ \forall\ (i,j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\}\rangle,$$

where $f = \mathbf{f}(\mathsf{op})$.

The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- If $\mathsf{accum} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

  $$\widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i,j,Z_{ij}) \forall (i,j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

  $$Z_{ij} = \widetilde{\mathbf{C}}(i,j) \odot \widetilde{\mathbf{T}}(i,j),\ \text{if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}})),$$

  $$Z_{ij} = \widetilde{\mathbf{C}}(i,j),\ \text{if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

  $$Z_{ij} = \widetilde{\mathbf{T}}(i,j),\ \text{if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up matrix $\widetilde{\mathbf{Z}}$ are written into the final result matrix C, using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in C on input to this operation are deleted and the content of the new output matrix, C, is defined as,

$$\mathbf{L}(\mathsf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{Z}}$ indicated by the mask are copied into the result matrix, C, and elements of C that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathsf{C}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.9 reduce: Perform a reduction across the elements of an object

Computes the reduction of the values of the elements of a vector or matrix.

4.3.9.1 reduce: Standard matrix to vector variant

This performs a reduction across rows of a matrix to produce a vector. If column reduction across columns is desired, the input matrix should be transposed which can be specified using the descriptor.

C Syntax

```
GrB_Info GrB_reduce(GrB_Vector             w,
                    const GrB_Vector       mask,
                    const GrB_BinaryOp     accum,
                    const GrB_Monoid       op,
                    const GrB_Matrix       A,
                    const GrB_Descriptor   desc);

GrB_Info GrB_reduce(GrB_Vector             w,
                    const GrB_Vector       mask,
                    const GrB_BinaryOp     accum,
                    const GrB_BinaryOp     op,
                    const GrB_Matrix       A,
                    const GrB_Descriptor   desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the reduction operation. On output, this vector holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w and the domain of the mask vector must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default vector is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) The monoid or binary operator used in the element-wise reduction operation. Depending on which type is passed, the following defines the binary operator with one domain, $F_b = \langle D, D, D, \oplus \rangle$, that is used:

BinaryOp: $F_b = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{D}_{in_1}(\mathsf{op}), \mathbf{D}_{in_2}(\mathsf{op}), \bigodot(\mathsf{op}) \rangle$.

Monoid: $F_b = \langle \mathbf{D}(\mathsf{op}), \mathbf{D}(\mathsf{op}), \mathbf{D}(\mathsf{op}), \bigodot(\mathsf{op}) \rangle$, the identity element of the monoid is ignored.

If op is a GrB_BinaryOp, then all its domains must be the same. Furthermore, in both cases $\bigodot(\mathsf{op})$ must be commutative and associative. Otherwise, the outcome of the operation is undefined.

A (IN) The GraphBLAS matrix on which reduction will be performed.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_SCMP	Use the structural complement of mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for vector parameters).

GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

GrB_DOMAIN_MISMATCH Either the domains of the various vectors and matrices are incompatible with the corresponding domains of the accumulating operation, mask, and reduce function, or the domains of the GraphBLAS binary operator op are not all the same.

Description

This variant of GrB_reduce computes the result of performing a reduction across each of the rows of an input matrix: $\mathsf{w}(i) = \bigoplus \mathsf{A}(i,:) \forall i$; or, if an optional binary accumulation operator is provided, $\mathsf{w}(i) = \mathsf{w}(i) \odot (\bigoplus \mathsf{A}(i,:)) \forall i$, where $\bigoplus = \bigodot(F_b)$ and $\odot = \bigodot(\mathsf{accum})$.

Logically, this operation occurs in three steps:

Setup The internal vector, matrix and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to two vector and one matrix argument are used in this GrB_reduce operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\} \rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\} \rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\} \rangle$

The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{w})$ must be compatible with the domain of the reduction binary operator, $\mathbf{D}(F_b)$.

3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(F_b)$, must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

4. $\mathbf{D}(\mathsf{A})$ must be compatible with the domain of the binary reduction operator, $\mathbf{D}(F_b)$.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_reduce ends and the domain mismatch error listed above is returned.

From the argument vectors, the internal vectors and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.

2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:

   (a) If $\mathsf{mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathsf{w})\} \rangle$.

   (b) Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\} \rangle$.

   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$.

3. Matrix $\widetilde{\mathbf{A}} \leftarrow$ desc[GrB_INP0].GrB_TRAN ? A^T : A.

The internal vectors and masks are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}})$

2. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

If any compatibility rule above is violated, execution of GrB_reduce ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We carry out the reduce and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the result from reducing along the rows of input matrix $\widetilde{\mathbf{A}}$.

- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector, $\widetilde{\mathbf{t}}$, is created as follows:

$$\widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathsf{op}), \mathbf{size}(\widetilde{\mathbf{w}}), \mathbf{L}(\widetilde{\mathbf{t}}) = \{(i, t_i) : \mathbf{ind}(A(i,:)) \neq \emptyset\} \rangle.$$

The value of each of its elements is computed by

$$t_i = \bigoplus_{j \in \mathbf{ind}(\widetilde{\mathbf{A}}(i,:))} \widetilde{\mathbf{A}}(i,j),$$

where $\bigoplus = \bigodot(F_b)$.

The intermediate vector $\widetilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- If accum = GrB_NULL, then $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{z}}$ is defined as

$$\widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i)\ \forall\ i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{w}}$ and $\widetilde{\mathbf{t}}$.

$$z_i = \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i),\ \text{if}\ i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})),$$

$$z_i = \widetilde{\mathbf{w}}(i),\ \text{if}\ i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

$$z_i = \widetilde{\mathbf{t}}(i),\ \text{if}\ i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up vector $\widetilde{\mathbf{z}}$ are written into the final result vector w, using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in w on input to this operation are deleted and the content of the new output vector, w, is defined as,

$$\mathbf{L}(\mathsf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{z}}$ indicated by the mask are copied into the result vector, w, and elements of w that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathsf{w}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\widetilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.3.9.2 reduce: Vector-scalar variant

Reduce all stored values into a single scalar.

C Syntax

```
GrB_Info GrB_reduce(<type>                *val,
                    const GrB_BinaryOp    accum,
                    const GrB_Monoid      op,
                    const GrB_Vector      u,
                    const GrB_Descriptor  desc);
```

Parameters

val (INOUT) Scalar to store final reduced value into. On input, the scalar provides a value that may be accumulated with the result of the reduction operation. On output, this scalar holds the results of the operation.

accum (IN) An optional binary operator used for accumulating entries into existing val value. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) The monoid used in the element-wise reduction operation, $M = \langle D, \oplus, 0 \rangle$. The binary operator, $\oplus$, must be commutative and associative; otherwise, the outcome of the operation is undefined.

u (IN) The GraphBLAS vector on which reduction will be performed.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description

Note: This argument is defined for consistency with the other GraphBLAS operations. There are currently no non-default field/value pairs that can be set for this operation.

Return Values

GrB_SUCCESS In blocking or non-blocking mode, the operation completed successfully, and the output scalar val is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized by a call to new (or Vector_dup for vector parameters).
GrB_DOMAIN_MISMATCH	The domains of input and output arguments are incompatible with the corresponding domains of the accumulation operator, or reduce operator.
GrB_NULL_POINTER	val pointer is NULL.

Description

This variant of GrB_reduce computes the result of performing a reduction across each of the elements of an input vector: $\mathsf{val} = \bigoplus \mathsf{u}(:)$; or, if an optional binary accumulation operator is provided, $\mathsf{val} = \mathsf{val} \odot (\bigoplus \mathsf{u}(:))$, where $\bigoplus = \bigodot(\mathsf{op})$ and $\odot = \bigodot(\mathsf{accum})$.

Logically, this operation occurs in three steps:

Setup The internal vector used in the computation is formed and its domain is tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output scalar.

One vector argument is used in this GrB_reduce operation:

1. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i, u_i)\} \rangle$

The output scalar, argument vector, reduction operator and accumulation operator (if provided) are tested for domain compatibility as follows:

1. If accum is GrB_NULL, then $\mathbf{D}(\mathsf{val})$ must be compatible with $\mathbf{D}(\mathsf{op})$ of the reduction binary operator.

2. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{val})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{op})$ of the reduction binary operator must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

3. $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}(\mathsf{op})$ of the binary reduction operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_reduce ends and the domain mismatch error listed above is returned.

From the argument vector, the internal vector used in the computation is formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.

We are now ready to carry out the reduce and any additional associated operations. First, an intermediate scalar result t is computed using the recurrence:

$$t \leftarrow \mathbf{0}(\mathsf{op}),$$

$$t \leftarrow t \oplus \mathbf{u}(i), \forall i \in \mathbf{ind}(\mathbf{u}).$$

Where $\oplus = \bigodot(\mathsf{op})$, and $\mathbf{0}(\mathsf{op})$ is the identity of the monoid.

The final reduction value val is computed as follows:

- If accum = GrB_NULL, then val $\leftarrow t$.
- If accum is a binary operator, then val $\leftarrow$ val $\odot\, t$, where $\odot = \bigodot(\mathsf{accum})$.

In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value GrB_SUCCESS and the new contents of val is as defined above and fully computed.

4.3.9.3 reduce: Matrix-scalar variant

Reduce all stored values into a single scalar.

C Syntax

```
GrB_Info GrB_reduce(<type>              *val,
                    const GrB_BinaryOp   accum,
                    const GrB_Monoid     op,
                    const GrB_Matrix     A,
                    const GrB_Descriptor desc);
```

Parameters

val (INOUT) Scalar to store final reduced value into. On input, the scalar provides a value that may be accumulated with the result of the reduction operation. On output, this scalar holds the results of the operation.

accum (IN) An optional binary operator used for accumulating entries into existing val value. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) The monoid used in the element-wise reduction operation, $M = \langle D, \oplus, 0 \rangle$. The binary operator, $\oplus$, must be commutative and associative; otherwise, the outcome of the operation is undefined.

A (IN) The GraphBLAS matrix on which reduction will be performed.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description

Note: This argument is defined for consistency with the other GraphBLAS operations. There are currently no non-default field/value pairs that can be set for this operation.

Return Values

GrB_SUCCESS In blocking or non-blocking mode, the operation completed successfully, and the output scalar val is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with the corresponding domains of the accumulation operator, or reduce operator.

GrB_NULL_POINTER val pointer is NULL.

Description

This variant of GrB_reduce computes the result of performing a reduction across each of the elements of an input matrix: $\mathsf{val} = \bigoplus \mathsf{A}(:,:)$; or, if an optional binary accumulation operator is provided, $\mathsf{val} = \mathsf{val} \odot (\bigoplus \mathsf{A}(:,:))$, where $\bigoplus = \bigodot(\mathsf{op})$ and $\odot = \bigodot(\mathsf{accum})$.

Logically, this operation occurs in three steps:

Setup The internal matrix used in the computation is formed and its domain is tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output scalar.

One matrix argument is used in this GrB_reduce operation:

1. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{size}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{i,j})\}\rangle$

The output scalar, argument matrix, reduction operator and accumulation operator (if provided) are tested for domain compatibility as follows:

1. If accum is GrB_NULL, then $\mathbf{D}(\mathsf{val})$ must be compatible with $\mathbf{D}(\mathsf{op})$ of the reduction binary operator.
2. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{val})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{op})$ of the reduction binary operator must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.
3. $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}(\mathsf{op})$ of the binary reduction operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_reduce ends and the domain mismatch error listed above is returned.

From the argument matrix, the internal matrix used in the computation is formed ($\leftarrow$ denotes copy):

1. Matrix $\widetilde{\mathbf{A}} \leftarrow \mathsf{A}$.

We are now ready to carry out the reduce and any additional associated operations. First, an intermediate scalar result t is computed using the recurrence:

$$t \leftarrow \mathbf{0}(\mathsf{op}),$$

$$t \leftarrow t \oplus \mathbf{A}(i,j), \forall (i,j) \in \mathbf{ind}(\mathbf{A}).$$

Where $\oplus = \bigodot(\mathsf{op})$, and $\mathbf{0}(\mathsf{op})$ is the identity of the monoid.

The final reduction value val is computed as follows:

- If $\mathsf{accum} = \mathsf{GrB\_NULL}$, then $\mathsf{val} \leftarrow t$.
- If accum is a binary operator, then $\mathsf{val} \leftarrow \mathsf{val} \odot t$, where $\odot = \bigodot(\mathsf{accum})$.

In both GrB_BLOCKING and GrB_NONBLOCKING modes, the method exits with return value GrB_SUCCESS and the new contents of val is as defined above and fully computed.

4.3.10 transpose: Transpose rows and columns of a matrix

This version computes a new matrix that is the transpose of the source matrix.

C Syntax

```
GrB_Info GrB_transpose(GrB_Matrix            C,
                       const GrB_Matrix       Mask,
                       const GrB_BinaryOp     accum,
                       const GrB_Matrix       A,
                       const GrB_Descriptor   desc);
```

Parameters

- **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the transpose operation. On output, the matrix holds the results of the operation.
- **Mask** (IN) An optional "write" mask that controls which results from this operation are stored into the output matrix C. The mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined "built-in" types in Table 2.2. If the default matrix is desired (i.e., with correct dimensions and filled with true), GrB_NULL should be specified.
- **accum** (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.
- **A** (IN) The GraphBLAS matrix on which transposition will be performed.
- **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_SCMP	Use the structural complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

Return Values

- **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.
- **GrB_PANIC** Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

GrB_DIMENSION_MISMATCH mask, C and/or A dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool.

Description

GrB_transpose computes the result of performing a transpose of the input matrix: $\mathsf{C} = \mathsf{A}^T$; or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C} = \mathsf{C} \odot \mathsf{A}^T$. We note that the input matrix A can itself be optionally transposed before the operation, which would cause either an assignment from A to C or an accumulation of A into C.

Logically, this operation occurs in three steps:

Setup The internal matrix and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to three matrix arguments are used in this GrB_transpose operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i,j,C_{ij})\} \rangle$
2. $\mathsf{Mask} = \langle \mathbf{D}(\mathsf{Mask}), \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \mathbf{L}(\mathsf{Mask}) = \{(i,j,M_{ij})\} \rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i,j,A_{ij})\} \rangle$

The argument matrices and accumulation operator (if provided) are tested for domain compatibility as follows:

1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
2. $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}(\mathsf{A})$ of the input matrix.
3. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{C})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}(\mathsf{A})$ of the input matrix must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_transpose ends and the domain mismatch error listed above is returned.

From the argument matrices, the internal matrices and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Matrix $\widetilde{\mathbf{C}} \leftarrow \mathsf{C}$.
2. Two-dimensional mask, $\widetilde{\mathbf{M}}$, is computed from argument Mask as follows:
   (a) If $\mathsf{Mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \{(i,j), \forall i,j : 0 \leq i < \mathbf{nrows}(\mathsf{C}), 0 \leq j < \mathbf{ncols}(\mathsf{C})\}\rangle$.
   (b) Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \{(i,j) : (i,j) \in \mathbf{ind}(\mathsf{Mask}) \wedge (\mathsf{bool})\mathsf{Mask}(i,j) = \mathsf{true}\}\rangle$.
   (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}$.
3. Matrix $\widetilde{\mathbf{A}} \leftarrow$ desc[GrB_INP0].GrB_TRAN ? $\mathsf{A}^T : \mathsf{A}$.

The internal matrices and masks are checked for dimension compatibility. The following conditions must hold:

1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.
4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

If any compatibility rule above is violated, execution of GrB_transpose ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the matrix transposition and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\widetilde{\mathbf{T}}$: The matrix holding the transpose of $\widetilde{\mathbf{A}}$.
- $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

The intermediate matrix

$$\widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathsf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{L}(\widetilde{\mathbf{T}}) = \{(j, i, A_{ij}) \forall (i,j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\rangle$$

is created.

The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- If $\mathsf{accum} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$\widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathsf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i,j,Z_{ij}) \forall (i,j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\}\rangle.$$

  The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j) \odot \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}})),$$

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

$$Z_{ij} = \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

  where $\odot = \bigodot(\mathsf{accum})$, and the difference operator refers to set difference.

Finally, the set of output values that make up matrix $\widetilde{\mathbf{Z}}$ are written into the final result matrix C, using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a "write mask".

- If desc[GrB_OUTP].GrB_REPLACE is set, then any values in C on input to this operation are deleted and the content of the new output matrix, C, is defined as,

$$\mathbf{L}(\mathsf{C}) = \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

- If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\widetilde{\mathbf{Z}}$ indicated by the mask are copied into the result matrix, C, and elements of C that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathsf{C}) = \{(i,j,C_{ij}) : (i,j) \in (\mathbf{ind}(\mathsf{C}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{M}}))\} \cup \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$$

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix C is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4.4 Sequence Termination

4.4.1 wait: Waits until pending operations complete

When running in non-blocking mode, this function guarantees that all pending GraphBLAS operations are fully executed. Note that this can be called in blocking mode without an error, but there should be no pending GraphBLAS operations to complete.

C Syntax

```
GrB_Info GrB_wait();
```

Parameters

Return values

GrB_SUCCESS	operation completed successfully.
GrB_INDEX_OUT_OF_BOUNDS	an index out-of-bounds execution error happend during completion of pending operations.
GrB_OUT_OF_MEMORY	and out-of-memory execution error happened during completion of pending operations.
GrB_PANIC	unknown internal error.

Description

Upon successful return, all previously called GraphBLAS methods have fully completed their execution, and any (transparent or opaque) data structures produced or manipulated by those methods can be safely touched. If an error occured in any pending GraphBLAS operations, GrB_error() can be used to retrieve implementation defined error information about the problem encountered.

4.4.2 error: Get an error message regarding internal errors

```
const char *GrB_error();
```

Parameters

Return value

- A pointer to a null-terminated string (owned by the library).

Description

After a call to any GraphBLAS method, the program can retrieve additional error information (beyond the error code returned by the method) though a call to the function GrB_error(). The function returns a pointer to a null terminated string and the contents of that string are implementation dependent. In particular, a null string (not a NULL pointer) is always a valid error string. The pointer is valid until the next call to any GraphBLAS method by the same thread. GrB_error() is a thread-safe function, in the sense that multiple threads can call it simultaneously and each will get its own error string back, referring to the last GraphBLAS method it called.

Chapter 5

Nonpolymorphic Interface

Each polymorphic GraphBLAS method (those with multiple parameter signatures under the same name) has a corresponding set of long-name forms that are specific to each parameter signature. That is show in Tables 5.1 through 5.6.

Table 5.1: Long-name, nonpolymorphic form of GraphBLAS methods.

Polymorphic signature	Nonpolymorphic signature
GrB_Monoid_new(GrB_Monoid*,...,bool)	GrB_Monoid_new_BOOL(GrB_Monoid*,GrB_BinaryOp,bool)
GrB_Monoid_new(GrB_Monoid*,...,int8_t)	GrB_Monoid_new_INT8(GrB_Monoid*,GrB_BinaryOp,int8_t)
GrB_Monoid_new(GrB_Monoid*,...,uint8_t)	GrB_Monoid_new_UINT8(GrB_Monoid*,GrB_BinaryOp,uint8_t)
GrB_Monoid_new(GrB_Monoid*,...,int16_t)	GrB_Monoid_new_INT16(GrB_Monoid*,GrB_BinaryOp,int16_t)
GrB_Monoid_new(GrB_Monoid*,...,uint16_t)	GrB_Monoid_new_UINT16(GrB_Monoid*,GrB_BinaryOp,uint16_t)
GrB_Monoid_new(GrB_Monoid*,...,int32_t)	GrB_Monoid_new_INT32(GrB_Monoid*,GrB_BinaryOp,int32_t)
GrB_Monoid_new(GrB_Monoid*,...,uint32_t)	GrB_Monoid_new_UINT32(GrB_Monoid*,GrB_BinaryOp,uint32_t)
GrB_Monoid_new(GrB_Monoid*,...,int64_t)	GrB_Monoid_new_INT64(GrB_Monoid*,GrB_BinaryOp,int64_t)
GrB_Monoid_new(GrB_Monoid*,...,uint64_t)	GrB_Monoid_new_UINT64(GrB_Monoid*,GrB_BinaryOp,uint64_t)
GrB_Monoid_new(GrB_Monoid*,...,float)	GrB_Monoid_new_FP32(GrB_Monoid*,GrB_BinaryOp,float)
GrB_Monoid_new(GrB_Monoid*,...,double)	GrB_Monoid_new_FP64(GrB_Monoid*,GrB_BinaryOp,double)
GrB_Monoid_new(GrB_Monoid*,...,*other*)	GrB_Monoid_new_UDT(GrB_Monoid*,GrB_BinaryOp,void*)

Table 5.2: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Vector_build(...,const bool*,...)	GrB_Vector_build_BOOL(...,const bool*,...)
GrB_Vector_build(...,const int8_t*,...)	GrB_Vector_build_INT8(...,const int8_t*,...)
GrB_Vector_build(...,const uint8_t*,...)	GrB_Vector_build_UINT8(...,const uint8_t*,...)
GrB_Vector_build(...,const int16_t*,...)	GrB_Vector_build_INT16(...,const int16_t*,...)
GrB_Vector_build(...,const uint16_t*,...)	GrB_Vector_build_UINT16(...,const uint16_t*,...)
GrB_Vector_build(...,const int32_t*,...)	GrB_Vector_build_INT32(...,const int32_t*,...)
GrB_Vector_build(...,const uint32_t*,...)	GrB_Vector_build_UINT32(...,const uint32_t*,...)
GrB_Vector_build(...,const int64_t*,...)	GrB_Vector_build_INT64(...,const int64_t*,...)
GrB_Vector_build(...,const uint64_t*,...)	GrB_Vector_build_UINT64(...,const uint64_t*,...)
GrB_Vector_build(...,const float*,...)	GrB_Vector_build_FP32(...,const float*,...)
GrB_Vector_build(...,const double*,...)	GrB_Vector_build_FP64(...,const double*,...)
GrB_Vector_build(...,*other*,...)	GrB_Vector_build_UDT(...,const void*,...)
GrB_Vector_setElement(..., bool,...)	GrB_Vector_setElement_BOOL(..., bool,...)
GrB_Vector_setElement(..., int8_t,...)	GrB_Vector_setElement_INT8(..., int8_t,...)
GrB_Vector_setElement(..., uint8_t,...)	GrB_Vector_setElement_UINT8(..., uint8_t,...)
GrB_Vector_setElement(..., int16_t,...)	GrB_Vector_setElement_INT16(..., int16_t,...)
GrB_Vector_setElement(..., uint16_t,...)	GrB_Vector_setElement_UINT16(..., uint16_t,...)
GrB_Vector_setElement(..., int32_t,...)	GrB_Vector_setElement_INT32(..., int32_t,...)
GrB_Vector_setElement(..., uint32_t,...)	GrB_Vector_setElement_UINT32(..., uint32_t,...)
GrB_Vector_setElement(..., int64_t,...)	GrB_Vector_setElement_INT64(..., int64_t,...)
GrB_Vector_setElement(..., uint64_t,...)	GrB_Vector_setElement_UINT64(..., uint64_t,...)
GrB_Vector_setElement(..., float,...)	GrB_Vector_setElement_FP32(..., float,...)
GrB_Vector_setElement(..., double,...)	GrB_Vector_setElement_FP64(..., double,...)
GrB_Vector_setElement(...,*other*,...)	GrB_Vector_setElement_UDT(...,const void*,...)
GrB_Vector_extractElement(bool*,...)	GrB_Vector_extractElement_BOOL(bool*,...)
GrB_Vector_extractElement(int8_t*,...)	GrB_Vector_extractElement_INT8(int8_t*,...)
GrB_Vector_extractElement(uint8_t*,...)	GrB_Vector_extractElement_UINT8(uint8_t*,...)
GrB_Vector_extractElement(int16_t*,...)	GrB_Vector_extractElement_INT16(int16_t*,...)
GrB_Vector_extractElement(uint16_t*,...)	GrB_Vector_extractElement_UINT16(uint16_t*,...)
GrB_Vector_extractElement(int32_t*,...)	GrB_Vector_extractElement_INT32(int32_t*,...)
GrB_Vector_extractElement(uint32_t*,...)	GrB_Vector_extractElement_UINT32(uint32_t*,...)
GrB_Vector_extractElement(int64_t*,...)	GrB_Vector_extractElement_INT64(int64_t*,...)
GrB_Vector_extractElement(uint64_t*,...)	GrB_Vector_extractElement_UINT64(uint64_t*,...)
GrB_Vector_extractElement(float*,...)	GrB_Vector_extractElement_FP32(float*,...)
GrB_Vector_extractElement(double*,...)	GrB_Vector_extractElement_FP64(double*,...)
GrB_Vector_extractElement(*other*,...)	GrB_Vector_extractElement_UDT(void*,...)
GrB_Vector_extractTuples(..., bool*,...)	GrB_Vector_extractTuples_BOOL(..., bool*,...)
GrB_Vector_extractTuples(..., int8_t*,...)	GrB_Vector_extractTuples_INT8(..., int8_t*,...)
GrB_Vector_extractTuples(..., uint8_t*,...)	GrB_Vector_extractTuples_UINT8(..., uint8_t*,...)
GrB_Vector_extractTuples(..., int16_t*,...)	GrB_Vector_extractTuples_INT16(..., int16_t*,...)
GrB_Vector_extractTuples(..., uint16_t*,...)	GrB_Vector_extractTuples_UINT16(..., uint16_t*,...)
GrB_Vector_extractTuples(..., int32_t*,...)	GrB_Vector_extractTuples_INT32(..., int32_t*,...)
GrB_Vector_extractTuples(..., uint32_t*,...)	GrB_Vector_extractTuples_UINT32(..., uint32_t*,...)
GrB_Vector_extractTuples(..., int64_t*,...)	GrB_Vector_extractTuples_INT64(..., int64_t*,...)
GrB_Vector_extractTuples(..., uint64_t*,...)	GrB_Vector_extractTuples_UINT64(..., uint64_t*,...)
GrB_Vector_extractTuples(..., float*,...)	GrB_Vector_extractTuples_FP32(..., float*,...)
GrB_Vector_extractTuples(..., double*,...)	GrB_Vector_extractTuples_FP64(..., double*,...)
GrB_Vector_extractTuples(...,*other*,...)	GrB_Vector_extractTuples_UDT(..., void*,...)

Table 5.3: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_Matrix_build(...,const bool*,...)	GrB_Matrix_build_BOOL(...,const bool*,...)
GrB_Matrix_build(...,const int8_t*,...)	GrB_Matrix_build_INT8(...,const int8_t*,...)
GrB_Matrix_build(...,const uint8_t*,...)	GrB_Matrix_build_UINT8(...,const uint8_t*,...)
GrB_Matrix_build(...,const int16_t*,...)	GrB_Matrix_build_INT16(...,const int16_t*,...)
GrB_Matrix_build(...,const uint16_t*,...)	GrB_Matrix_build_UINT16(...,const uint16_t*,...)
GrB_Matrix_build(...,const int32_t*,...)	GrB_Matrix_build_INT32(...,const int32_t*,...)
GrB_Matrix_build(...,const uint32_t*,...)	GrB_Matrix_build_UINT32(...,const uint32_t*,...)
GrB_Matrix_build(...,const int64_t*,...)	GrB_Matrix_build_INT64(...,const int64_t*,...)
GrB_Matrix_build(...,const uint64_t*,...)	GrB_Matrix_build_UINT64(...,const uint64_t*,...)
GrB_Matrix_build(...,const float*,...)	GrB_Matrix_build_FP32(...,const float*,...)
GrB_Matrix_build(...,const double*,...)	GrB_Matrix_build_FP64(...,const double*,...)
GrB_Matrix_build(...,*other*,...)	GrB_Matrix_build_UDT(...,const void*,...)
GrB_Matrix_setElement(..., bool,...)	GrB_Matrix_setElement_BOOL(..., bool,...)
GrB_Matrix_setElement(..., int8_t,...)	GrB_Matrix_setElement_INT8(..., int8_t,...)
GrB_Matrix_setElement(..., uint8_t,...)	GrB_Matrix_setElement_UINT8(..., uint8_t,...)
GrB_Matrix_setElement(..., int16_t,...)	GrB_Matrix_setElement_INT16(..., int16_t,...)
GrB_Matrix_setElement(..., uint16_t,...)	GrB_Matrix_setElement_UINT16(..., uint16_t,...)
GrB_Matrix_setElement(..., int32_t,...)	GrB_Matrix_setElement_INT32(..., int32_t,...)
GrB_Matrix_setElement(..., uint32_t,...)	GrB_Matrix_setElement_UINT32(..., uint32_t,...)
GrB_Matrix_setElement(..., int64_t,...)	GrB_Matrix_setElement_INT64(..., int64_t,...)
GrB_Matrix_setElement(..., uint64_t,...)	GrB_Matrix_setElement_UINT64(..., uint64_t,...)
GrB_Matrix_setElement(..., float,...)	GrB_Matrix_setElement_FP32(..., float,...)
GrB_Matrix_setElement(..., double,...)	GrB_Matrix_setElement_FP64(..., double,...)
GrB_Matrix_setElement(...,*other*,...)	GrB_Matrix_setElement_UDT(...,const void*,...)
GrB_Matrix_extractElement(bool*,...)	GrB_Matrix_extractElement_BOOL(bool*,...)
GrB_Matrix_extractElement(int8_t*,...)	GrB_Matrix_extractElement_INT8(int8_t*,...)
GrB_Matrix_extractElement(uint8_t*,...)	GrB_Matrix_extractElement_UINT8(uint8_t*,...)
GrB_Matrix_extractElement(int16_t*,...)	GrB_Matrix_extractElement_INT16(int16_t*,...)
GrB_Matrix_extractElement(uint16_t*,...)	GrB_Matrix_extractElement_UINT16(uint16_t*,...)
GrB_Matrix_extractElement(int32_t*,...)	GrB_Matrix_extractElement_INT32(int32_t*,...)
GrB_Matrix_extractElement(uint32_t*,...)	GrB_Matrix_extractElement_UINT32(uint32_t*,...)
GrB_Matrix_extractElement(int64_t*,...)	GrB_Matrix_extractElement_INT64(int64_t*,...)
GrB_Matrix_extractElement(uint64_t*,...)	GrB_Matrix_extractElement_UINT64(uint64_t*,...)
GrB_Matrix_extractElement(float*,...)	GrB_Matrix_extractElement_FP32(float*,...)
GrB_Matrix_extractElement(double*,...)	GrB_Matrix_extractElement_FP64(double*,...)
GrB_Matrix_extractElement(*other*,...)	GrB_Matrix_extractElement_UDT(void*,...)
GrB_Matrix_extractTuples(..., bool*,...)	GrB_Matrix_extractTuples_BOOL(..., bool*,...)
GrB_Matrix_extractTuples(..., int8_t*,...)	GrB_Matrix_extractTuples_INT8(..., int8_t*,...)
GrB_Matrix_extractTuples(..., uint8_t*,...)	GrB_Matrix_extractTuples_UINT8(..., uint8_t*,...)
GrB_Matrix_extractTuples(..., int16_t*,...)	GrB_Matrix_extractTuples_INT16(..., int16_t*,...)
GrB_Matrix_extractTuples(..., uint16_t*,...)	GrB_Matrix_extractTuples_UINT16(..., uint16_t*,...)
GrB_Matrix_extractTuples(..., int32_t*,...)	GrB_Matrix_extractTuples_INT32(..., int32_t*,...)
GrB_Matrix_extractTuples(..., uint32_t*,...)	GrB_Matrix_extractTuples_UINT32(..., uint32_t*,...)
GrB_Matrix_extractTuples(..., int64_t*,...)	GrB_Matrix_extractTuples_INT64(..., int64_t*,...)
GrB_Matrix_extractTuples(..., uint64_t*,...)	GrB_Matrix_extractTuples_UINT64(..., uint64_t*,...)
GrB_Matrix_extractTuples(..., float*,...)	GrB_Matrix_extractTuples_FP32(..., float*,...)
GrB_Matrix_extractTuples(..., double*,...)	GrB_Matrix_extractTuples_FP64(..., double*,...)
GrB_Matrix_extractTuples(...,*other*,...)	GrB_Matrix_extractTuples_UDT(..., void*,...)

Table 5.4: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_free(GrB_Type*)	GrB_Type_free(GrB_Type*)
GrB_free(GrB_UnaryOp*)	GrB_UnaryOp_free(GrB_UnaryOp*)
GrB_free(GrB_BinaryOp*)	GrB_BinaryOp_free(GrB_BinaryOp*)
GrB_free(GrB_Monoid*)	GrB_Monoid_free(GrB_Monoid*)
GrB_free(GrB_Semiring*)	GrB_Semiring_free(GrB_Semiring*)
GrB_free(GrB_Vector*)	GrB_Vector_free(GrB_Vector*)
GrB_free(GrB_Matrix*)	GrB_Matrix_free(GrB_Matrix*)
GrB_free(GrB_Descriptor*)	GrB_Descriptor_free(GrB_Descriptor*)
GrB_eWiseMult(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseMult_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseMult_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseMult_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseMult_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseMult_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseMult(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseMult_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Semiring,...)	GrB_Vector_eWiseAdd_Semiring(GrB_Vector,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_Monoid,...)	GrB_Vector_eWiseAdd_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Vector_eWiseAdd_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Semiring,...)	GrB_Matrix_eWiseAdd_Semiring(GrB_Matrix,...,GrB_Semiring,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_Monoid,...)	GrB_Matrix_eWiseAdd_Monoid(GrB_Matrix,...,GrB_Monoid,...)
GrB_eWiseAdd(GrB_Matrix,...,GrB_BinaryOp,...)	GrB_Matrix_eWiseAdd_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)

Table 5.5: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_extract(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_extract(GrB_Vector,...,GrB_Vector,...)
GrB_extract(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_extract(GrB_Matrix,...,GrB_Matrix,...)
GrB_extract(GrB_Vector,...,GrB_Matrix,...)	GrB_Col_extract(GrB_Vector,...,GrB_Matrix,...)
GrB_assign(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_assign(GrB_Vector,...,GrB_Vector,...)
GrB_assign(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_assign(GrB_Matrix,...,GrB_Matrix,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)	GrB_Col_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)	GrB_Row_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)
GrB_assign(GrB_Vector,..., bool,...)	GrB_Vector_assign_BOOL(GrB_Vector,..., bool,...)
GrB_assign(GrB_Vector,..., int8_t,...)	GrB_Vector_assign_INT8(GrB_Vector,..., int8_t,...)
GrB_assign(GrB_Vector,..., uint8_t,...)	GrB_Vector_assign_UINT8(GrB_Vector,..., uint8_t,...)
GrB_assign(GrB_Vector,..., int16_t,...)	GrB_Vector_assign_INT16(GrB_Vector,..., int16_t,...)
GrB_assign(GrB_Vector,..., uint16_t,...)	GrB_Vector_assign_UINT16(GrB_Vector,..., uint16_t,...)
GrB_assign(GrB_Vector,..., int32_t,...)	GrB_Vector_assign_INT32(GrB_Vector,..., int32_t,...)
GrB_assign(GrB_Vector,..., uint32_t,...)	GrB_Vector_assign_UINT32(GrB_Vector,..., uint32_t,...)
GrB_assign(GrB_Vector,..., int64_t,...)	GrB_Vector_assign_INT64(GrB_Vector,..., int64_t,...)
GrB_assign(GrB_Vector,..., uint64_t,...)	GrB_Vector_assign_UINT64(GrB_Vector,..., uint64_t,...)
GrB_assign(GrB_Vector,..., float,...)	GrB_Vector_assign_FP32(GrB_Vector,..., float,...)
GrB_assign(GrB_Vector,..., double,...)	GrB_Vector_assign_FP64(GrB_Vector,..., double,...)
GrB_assign(GrB_Vector,...,*other*,...)	GrB_Vector_assign_UDT(GrB_Vector,...,const void*,...)
GrB_assign(GrB_Matrix,..., bool,...)	GrB_Matrix_assign_BOOL(GrB_Matrix...., bool,...)
GrB_assign(GrB_Matrix,..., int8_t,...)	GrB_Matrix_assign_INT8(GrB_Matrix...., int8_t,...)
GrB_assign(GrB_Matrix,..., uint8_t,...)	GrB_Matrix_assign_UINT8(GrB_Matrix...., uint8_t,...)
GrB_assign(GrB_Matrix,..., int16_t,...)	GrB_Matrix_assign_INT16(GrB_Matrix...., int16_t,...)
GrB_assign(GrB_Matrix,..., uint16_t,...)	GrB_Matrix_assign_UINT16(GrB_Matrix...., uint16_t,...)
GrB_assign(GrB_Matrix,..., int32_t,...)	GrB_Matrix_assign_INT32(GrB_Matrix...., int32_t,...)
GrB_assign(GrB_Matrix,..., uint32_t,...)	GrB_Matrix_assign_UINT32(GrB_Matrix...., uint32_t,...)
GrB_assign(GrB_Matrix,..., int64_t,...)	GrB_Matrix_assign_INT64(GrB_Matrix...., int64_t,...)
GrB_assign(GrB_Matrix,..., uint64_t,...)	GrB_Matrix_assign_UINT64(GrB_Matrix...., uint64_t,...)
GrB_assign(GrB_Matrix,..., float,...)	GrB_Matrix_assign_FP32(GrB_Matrix...., float,...)
GrB_assign(GrB_Matrix,..., double,...)	GrB_Matrix_assign_FP64(GrB_Matrix...., double,...)
GrB_assign(GrB_Matrix,...,*other*,...)	GrB_Matrix_assign_UDT(GrB_Matrix....,const void*,...)
GrB_apply(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_apply(GrB_Vector,...,GrB_Vector,...)
GrB_apply(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_apply(GrB_Matrix,...,GrB_Matrix,...)

Table 5.6: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_reduce(GrB_Vector,...,GrB_Monoid,...)	GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)
GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)	GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)
GrB_reduce(bool*,...,GrB_Vector,...)	GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)
GrB_reduce(int8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)
GrB_reduce(uint8_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)
GrB_reduce(int16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)
GrB_reduce(uint16_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)
GrB_reduce(int32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)
GrB_reduce(uint32_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)
GrB_reduce(int64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)
GrB_reduce(uint64_t*,...,GrB_Vector,...)	GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)
GrB_reduce(float*,...,GrB_Vector,...)	GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)
GrB_reduce(double*,...,GrB_Vector,...)	GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)
GrB_reduce(*other*,...,GrB_Vector,...)	GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)
GrB_reduce(bool*,...,GrB_Matrix,...)	GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)
GrB_reduce(int8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)
GrB_reduce(uint8_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)
GrB_reduce(int16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)
GrB_reduce(uint16_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)
GrB_reduce(int32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)
GrB_reduce(uint32_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)
GrB_reduce(int64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)
GrB_reduce(uint64_t*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)
GrB_reduce(float*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)
GrB_reduce(double*,...,GrB_Matrix,...)	GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)
GrB_reduce(*other*,...,GrB_Matrix,...)	GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)

Appendix A

Revision History

Changes in 1.2.0:

- Removed ”provisional” clause.

Changes in 1.1.0:

- Removed unnecessary const from nindices, nrows, and ncols parameters of both extract and assign operations.
- Signature of GrB_UnaryOp_new changed: order of input parameters changed.
- Signature of GrB_BinaryOp_new changed: order of input parameters changed.
- Signature of GrB_Monoid_new changed: removal of domain argument which is now inferred from the domains of the binary operator provided.
- Signature of GrB_Vector_extractTuples and GrB_Matrix_extractTuples to add an in/out argument, n, which indicates the size of the output arrays provided (in terms of number of elements, not number of bytes). Added new execution error, GrB_INSUFFICIENT_SPACE which is returned when the capacities of the output arrays are insufficient to hold all of the tuples.
- Changed GrB_Column_assign to GrB_Col_assign for consistency in non-polymorphic interface.
- Added replace flag (z) notation to Table 4.1.
- Updated the “Mathematical Description” of the assign operation in Table 4.1.
- Added triangle counting example.
- Added subsection headers for accumulate and mask/replace discussions in the Description sections of GraphBLAS operations when the respective text was the “standard” text (i.e., identical in a majority of the operations).
- Fixed typographical errors.

Changes in 1.0.2:

- Expanded the definitions of Vector_build and Matrix_build to conceptually use intermediate matrices and avoid casting issues in certain implementations.
- Fixed the bug in the GrB_assign definition. Elements of the output object are no longer being erased outside the assigned area.
- Changes non-polymorphic interface:
  - Renamed GrB_Row_extract to GrB_Col_extract.
  - Renamed GrB_Vector_reduce_BinaryOp to GrB_Matrix_reduce_BinaryOp.
  - Renamed GrB_Vector_reduce_Monoid to GrB_Matrix_reduce_Monoid.
- Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- Fixed numerous typographical errors.

Appendix B

Examples

B.1 Example: breadth-first search (BFS) in GraphBLAS

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "GraphBLAS.h"

/*
 * Given a boolean n x n adjacency matrix A and a source vertex s, performs a BFS traversal
 * of the graph and sets v[i] to the level in which vertex i is visited (v[s] == 1).
 * If i is not reacheable from s, then v[i] = 0. (Vector v should be empty on input.)
 */
GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
{
  GrB_Index n;
  GrB_Matrix_nrows(&n,A);                          // n = # of rows of A

  GrB_Vector_new(v,GrB_INT32,n);                   // Vector<int32_t> v(n)

  GrB_Vector q;                                    // vertices visited in each level
  GrB_Vector_new(&q,GrB_BOOL,n);                   // Vector<bool> q(n)
  GrB_Vector_setElement(q,(bool)true,s);           // q[s] = true, false everywhere else

  GrB_Monoid Lor;                                  // Logical-or monoid
  GrB_Monoid_new(&Lor,GrB_LOR,(bool)false);

  GrB_Semiring Boolean;                            // Boolean semiring
  GrB_Semiring_new(&Boolean,Lor,GrB_LAND);

  GrB_Descriptor desc;                             // Descriptor for vxm
  GrB_Descriptor_new(&desc);
  GrB_Descriptor_set(desc,GrB_MASK,GrB_SCMP);      // invert the mask
  GrB_Descriptor_set(desc,GrB_OUTP,GrB_REPLACE);// clear the output before assignment

  /*
   * BFS traversal and label the vertices.
   */
  int32_t d = 0;                                   // d = level in BFS traversal
  bool succ = false;                               // succ == true when some successor found
  do {
    ++d;                                           // next level (start with 1)
    GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL);    // v[q] = d
    GrB_vxm(q,*v,GrB_NULL,Boolean,q,A,desc);       // q[!v] = q ||.&& A ; finds all the
                                                   // unvisited successors from current q
    GrB_reduce(&succ,GrB_NULL,Lor,q,GrB_NULL);     // succ = ||(q)
  } while (succ);                                  // if there is no successor in q, we are done.

  GrB_free(&q);                                    // q vector no longer needed
  GrB_free(&Lor);                                  // Logical or monoid no longer needed
  GrB_free(&Boolean);                              // Boolean semiring no longer needed
  GrB_free(&desc);                                 // descriptor no longer needed

  return GrB_SUCCESS;
}
```

B.2 Example: BFS in GraphBLAS using apply

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "GraphBLAS.h"

int32_t level = 0;               // level = depth in BFS traversal, roots=1, unvisited=0
void return_level(void *out, const void *in) {
  bool element = *(bool*)in;
  *(int32_t*)out = level;
}

/*
 * Given a boolean n x n adjacency matrix A and a source vertex s, performs a BFS traversal
 * of the graph and sets v[i] to the level in which vertex i is visited (v[s] == 1).
 * If i is not reacheable from s, then v[i] = 0. (Vector v should be empty on input.)
 */
GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
{
  GrB_Index n;
  GrB_Matrix_nrows(&n,A);                          // n = # of rows of A

  GrB_Vector_new(v,GrB_INT32,n);                   // Vector<int32_t> v(n) = 0

  GrB_Vector q;                                    // vertices visited in each level
  GrB_Vector_new(&q,GrB_BOOL,n);                   // Vector<bool> q(n) = false
  GrB_Vector_setElement(q,(bool)true,s);           // q[s] = true, false everywhere else

  GrB_Monoid Lor;                                  // Logical-or monoid
  GrB_Monoid_new(&Lor,GrB_LOR,false);

  GrB_Semiring Boolean;                            // Boolean semiring
  GrB_Semiring_new(&Boolean,Lor,GrB_LAND);

  GrB_Descriptor desc;                             // Descriptor for vxm
  GrB_Descriptor_new(&desc);
  GrB_Descriptor_set(desc,GrB_MASK,GrB_SCMP);      // invert the mask
  GrB_Descriptor_set(desc,GrB_OUTP,GrB_REPLACE);// clear the output before assignment

  GrB_UnaryOp apply_level;
  GrB_UnaryOp_new(&apply_level,return_level,GrB_INT32,GrB_BOOL);

  /*
   * BFS traversal and label the vertices.
   */
  level = 0;
  GrB_Index nvals;
  do {
    ++level;                                       // next level (start with 1)
    GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,apply_level,q,GrB_NULL);  // v[q] = level
    GrB_vxm(q,*v,GrB_NULL,Boolean,q,A,desc);       // q[!v] = q ||.&& A ; finds all the
                                                   // unvisited successors from current q
    GrB_Vector_nvals(&nvals, q);
  } while (nvals);                                 // if there is no successor in q, we are done.

  GrB_free(&q);                                    // q vector no longer needed
  GrB_free(&Lor);                                  // Logical or monoid no longer needed
  GrB_free(&Boolean);                              // Boolean semiring no longer needed
  GrB_free(&desc);                                 // descriptor no longer needed

  return GrB_SUCCESS;
}
```

B.3 Example: betweenness centrality (BC) in GraphBLAS

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "GraphBLAS.h"

/*
 * Given a boolean n x n adjacency matrix A and a source vertex s,
 * compute the BC-metric vector delta, which should be empty on input.
 */
GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
{
  GrB_Index n;
  GrB_Matrix_nrows(&n,A);                         // n = # of vertices in graph

  GrB_Vector_new(delta,GrB_FP32,n);               // Vector<float> delta(n)

  GrB_Matrix sigma;                               // Matrix<int32_t> sigma(n,n)
  GrB_Matrix_new(&sigma,GrB_INT32,n,n);           // sigma[d,k] = #shortest paths to node k at level d

  GrB_Vector q;
  GrB_Vector_new(&q, GrB_INT32, n);               // Vector<int32_t> q(n) of path counts
  GrB_Vector_setElement(q,1,s);                   // q[s] = 1

  GrB_Vector p;                                   // Vector<int32_t> p(n) shortest path counts so far
  GrB_Vector_dup(&p, q);                          // p = q

  GrB_Monoid Int32Add;                            // Monoid <int32_t,+,0>
  GrB_Monoid_new(&Int32Add,GrB_PLUS_INT32,0);
  GrB_Semiring Int32AddMul;                       // Semiring <int32_t,int32_t,int32_t,+,*,0,1>
  GrB_Semiring_new(&Int32AddMul,Int32Add,GrB_TIMES_INT32);

  GrB_Descriptor desc;                            // Descriptor for vxm
  GrB_Descriptor_new(&desc);
  GrB_Descriptor_set(desc,GrB_MASK,GrB_SCMP);     // structural complement of the mask
  GrB_Descriptor_set(desc,GrB_OUTP,GrB_REPLACE);  // clear the output before assignment

  GrB_Descriptor tr1;                             // Transpose 1st input argument
  GrB_Descriptor_new(&tr1);
  GrB_Descriptor_set(tr1,GrB_INP0,GrB_TRAN);      // structural complement of the mask

  /*
   * BFS phase
   */
  int32_t d = 0;                                  // BFS level number
  int32_t sum = 0;                                // sum == 0 when BFS phase is complete
  do {
    GrB_assign(sigma,GrB_NULL,GrB_NULL,q,d,GrB_ALL,n,GrB_NULL); // sigma[d,:] = q
    GrB_vxm(q,p,GrB_NULL,Int32AddMul,q,A,desc);                 // q = # paths to nodes reachable
                                                                //    from current level
    GrB_eWiseAdd(p,GrB_NULL,GrB_NULL,Int32AddMul,p,q,GrB_NULL); // accumulate path counts on this level
    GrB_reduce(&sum,GrB_NULL,Int32Add,q,GrB_NULL);              // sum path counts at this level
    ++d;
  } while (sum);

  /*
   * BC computation phase
   * (t1,t2,t3,t4) are temporary vectors
   */
  GrB_Monoid FP32Add;                             // Monoid <float,float,float,+,0.0>
  GrB_Monoid_new(&FP32Add,GrB_PLUS_FP32,0.0f);

```

```c
  GrB_Monoid FP32Mul;                             // Monoid <float,float,float,*,1.0>
  GrB_Monoid_new(&FP32Mul,GrB_TIMES_FP32,1.0f);

  GrB_Semiring FP32AddMul;                        // Semiring <float,float,float,+,*,0.0,1.0>
  GrB_Semiring_new(&FP32AddMul,FP32Add,GrB_TIMES_FP32);

  GrB_Vector t1; GrB_Vector_new(&t1,GrB_FP32,n);
  GrB_Vector t2; GrB_Vector_new(&t2,GrB_FP32,n);
  GrB_Vector t3; GrB_Vector_new(&t3,GrB_FP32,n);
  GrB_Vector t4; GrB_Vector_new(&t4,GrB_FP32,n);
  for(int i=d-1; i>0; i--)
  {
    GrB_assign(t1,GrB_NULL,GrB_NULL,1.0f,GrB_ALL,n,GrB_NULL);             // t1 = 1+delta
    GrB_eWiseAdd(t1,GrB_NULL,GrB_NULL,FP32Add,t1,*delta,GrB_NULL);
    GrB_extract(t2,GrB_NULL,GrB_NULL,sigma,GrB_ALL,n,i,tr1);               // t2 = sigma[i,:]
    GrB_eWiseMult(t2,GrB_NULL,GrB_NULL,GrB_DIV_FP32,t1,t2,GrB_NULL);      // t2 = (1+delta)/sigma[i,:]
    GrB_mxv(t3,GrB_NULL,GrB_NULL,FP32AddMul,A,t2,GrB_NULL);               // add contributions made by
                                                                           //     successors of a node
    GrB_extract(t4,GrB_NULL,GrB_NULL,sigma,GrB_ALL,n,i-1,tr1);             // t4 = sigma[i-1,:]
    GrB_eWiseMult(t4,GrB_NULL,GrB_NULL,FP32Mul,t4,t3,GrB_NULL);           // t4 = sigma[i-1,:]*t3
    GrB_eWiseAdd(*delta,GrB_NULL,GrB_NULL,FP32Add,*delta,t4,GrB_NULL);   // accumulate into delta
  }

  GrB_free(&sigma);
  GrB_free(&q); GrB_free(&p);
  GrB_free(&Int32AddMul); GrB_free(&Int32Add); GrB_free(&FP32AddMul);
  GrB_free(&FP32Add); GrB_free(&FP32Mul);
  GrB_free(&desc);
  GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);

  return GrB_SUCCESS;
}
```

B.4 Example: batched BC in GraphBLAS

```
#include <stdlib.h>
#include "GraphBLAS.h"  // in addition to other required C headers

// Compute partial BC metric for a subset of source vertices, s, in graph A
GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
{
  GrB_Index n;
  GrB_Matrix_nrows(&n, A);                                 // n = # of vertices in graph
  GrB_Vector_new(delta,GrB_FP32,n);                        // Vector<float> delta(n)

  GrB_Monoid Int32Add;                                     // Monoid <int32_t,+,0>
  GrB_Monoid_new(&Int32Add,GrB_PLUS_INT32,0);
  GrB_Semiring Int32AddMul;                                // Semiring <int32_t,int32_t,int32_t,+,*,0>
  GrB_Semiring_new(&Int32AddMul,Int32Add,GrB_TIMES_INT32);

  // Descriptor for BFS phase mxm
  GrB_Descriptor desc_tsr;
  GrB_Descriptor_new(&desc_tsr);
  GrB_Descriptor_set(desc_tsr,GrB_INP0,GrB_TRAN);          // transpose the adjacency matrix
  GrB_Descriptor_set(desc_tsr,GrB_MASK,GrB_SCMP);          // complement the mask
  GrB_Descriptor_set(desc_tsr,GrB_OUTP,GrB_REPLACE);       // clear output before result is stored

  // index and value arrays needed to build numsp
  GrB_Index *i_nsver = (GrB_Index*)malloc(sizeof(GrB_Index)*nsver);
  int32_t   *ones    = (int32_t*)  malloc(sizeof(int32_t)*nsver);
  for(int i=0; i<nsver; ++i) {
    i_nsver[i] = i;
    ones[i] = 1;
  }

  // numsp: structure holds the number of shortest paths for each node and starting vertex
  // discovered so far.  Initialized to source vertices:  numsp[s[i],i]=1, i=[0,nsver)
  GrB_Matrix numsp;
  GrB_Matrix_new( , GrB_INT32, n, nsver);
  GrB_Matrix_build(numsp,s,i_nsver,ones,nsver,GrB_PLUS_INT32);
  free(i_nsver); free(ones);                               //

  // frontier: Holds the current frontier where values are path counts.
  // Initialized to out vertices of each source node in s.
  GrB_Matrix frontier;
  GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
  GrB_extract(frontier,numsp,GrB_NULL,A,GrB_ALL,n,s,nsver,desc_tsr); //

  // sigma: stores frontier information for each level of BFS phase.  The memory
  // for an entry in sigmas is only allocated within the do-while loop if needed
  GrB_Matrix *sigmas = (GrB_Matrix*)malloc(sizeof(GrB_Matrix)*n);   // n is an upper bound on diameter

  int32_t   d = 0;                                         // BFS level number
  GrB_Index nvals = 0;                                     // nvals == 0 when BFS phase is complete

  // -------------------- The BFS phase (forward sweep) ---------------------------
  do {
    // sigmas[d](:,s) = d^th level frontier from source vertex s
    GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);

    GrB_apply(sigmas[d],GrB_NULL,GrB_NULL,
              GrB_IDENTITY_BOOL,frontier,GrB_NULL);        // sigmas[d](:,:) = (Boolean) frontier
    GrB_eWiseAdd(numsp,GrB_NULL,GrB_NULL,
                 Int32Add,numsp,frontier,GrB_NULL);        // numsp += frontier (accum path counts)
    GrB_mxm(frontier,numsp,GrB_NULL,
            Int32AddMul,A,frontier,desc_tsr);              // f<!numsp> = A' +.* f (update frontier)
    GrB_Matrix_nvals(&nvals,frontier);                     // number of nodes in frontier at this level
```

```
      d++;
   } while (nvals);

   GrB_Monoid FP32Add;                                        // Monoid <float,+,0.0>
   GrB_Monoid_new(&FP32Add,GrB_PLUS_FP32,0.0f);
   GrB_Monoid FP32Mul;                                        // Monoid <float,*,1.0>
   GrB_Monoid_new(&FP32Mul,GrB_TIMES_FP32,1.0f);
   GrB_Semiring FP32AddMul;                                   // Semiring <float,float,float,+,*,0.0>
   GrB_Semiring_new(&FP32AddMul,FP32Add,GrB_TIMES_FP32);

   // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
   GrB_Matrix nspinv;
   GrB_Matrix_new(&nspinv,GrB_FP32,n,nsver);
   GrB_apply(nspinv,GrB_NULL,GrB_NULL,
             GrB_MINV_FP32,numsp,GrB_NULL);                   // nspinv = 1./numsp

   // bcu: BC updates for each vertex for each starting vertex in s
   GrB_Matrix bcu;
   GrB_Matrix_new(&bcu,GrB_FP32,n,nsver);
   GrB_assign(bcu,GrB_NULL,GrB_NULL,
              1.0f,GrB_ALL,n, GrB_ALL,nsver,GrB_NULL);        // filled with 1 to avoid sparsity issues

   // Descriptor used in the tally phase
   GrB_Descriptor desc_r;
   GrB_Descriptor_new(&desc_r);
   GrB_Descriptor_set(desc_r,GrB_OUTP,GrB_REPLACE);           // clear output before result is stored

   GrB_Matrix w;                                              // temporary workspace matrix
   GrB_Matrix_new(&w,GrB_FP32,n,nsver);

   // ------------------------ Tally phase (backward sweep) ------------------------
   for (int i=d-1; i>0; i--) {
      GrB_eWiseMult(w,sigmas[i],GrB_NULL,
                    FP32Mul,bcu,nspinv,desc_r);               // w<sigmas[i]>=(1 ./ nsp).*bcu

      // add contributions by successors and mask with that BFS level's frontier
      GrB_mxm(w,sigmas[i-1],GrB_NULL,
              FP32AddMul,A,w,desc_r);                         // w<sigmas[i-1]> = (A +.* w)
      GrB_eWiseMult(bcu,GrB_NULL,GrB_PLUS_FP32,
                    FP32Mul,w,numsp,GrB_NULL);                // bcu += w .* numsp
   }

   // subtract "nsver" from every entry in delta (account for 1 extra value per bcu element)
   GrB_assign(*delta,GrB_NULL,GrB_NULL,
              -(float)nsver,GrB_ALL,n,GrB_NULL);              // fill with -nsver
   GrB_reduce(*delta,GrB_NULL,GrB_PLUS_FP32,
              GrB_PLUS_FP32,bcu,GrB_NULL);                    // add all updates to -nsver

   // Release resources
   for(int i=0; i<d; i++) {
      GrB_free(&(sigmas[i]));
   }
   free(sigmas);

   GrB_free(&frontier);      GrB_free( );
   GrB_free(&nspinv);        GrB_free(&bcu);        GrB_free(&w);
   GrB_free(&desc_tsr);      GrB_free(&desc_r);
   GrB_free(&Int32AddMul);   GrB_free(&Int32Add);
   GrB_free(&FP32AddMul);    GrB_free(&FP32Add);    GrB_free(&FP32Mul);

   return GrB_SUCCESS;
}
```

B.5 Example: maximal independent set (MIS) in GraphBLAS

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "GraphBLAS.h"

// Assign a random number to each element scaled by the inverse of the node's degree.
// This will increase the probability that low degree nodes are selected and larger
// sets are selected.
void setRandom(void *out, const void *in)
{
  uint32_t degree = *(uint32_t*)in;
  *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
}

/*
 * A variant of Luby's randomized algorithm [Luby 1985].
 *
 * Given a numeric n x n adjacency matrix A of an unwieghted and undirected graph (where
 * the value true represents an edge), compute a maximal set of independent vertices and
 * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
 * of the set (the iset vector should be uninitialized on input.)
 */
GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
{
  GrB_Index n;
  GrB_Matrix_nrows(&n,A);                        // n = # of rows of A

  GrB_Vector prob;                               // holds random probabilities for each node
  GrB_Vector neighbor_max;                       // holds value of max neighbor probability
  GrB_Vector new_members;                        // holds set of new members to iset
  GrB_Vector new_neighbors;                      // holds set of new neighbors to new iset mbrs.
  GrB_Vector candidates;                         // candidate members to iset

  GrB_Vector_new(&prob,GrB_FP32,n);
  GrB_Vector_new(&neighbor_max,GrB_FP32,n);
  GrB_Vector_new(&new_members,GrB_BOOL,n);
  GrB_Vector_new(&new_neighbors,GrB_BOOL,n);
  GrB_Vector_new(&candidates,GrB_BOOL,n);
  GrB_Vector_new(iset,GrB_BOOL,n);               // Initialize independent set vector, bool

  GrB_Monoid Max;
  GrB_Monoid_new(&Max,GrB_MAX_FP32,0.0f);

  GrB_Semiring maxSelect2nd;                     // Max/Select2nd "semiring"
  GrB_Semiring_new(&maxSelect2nd,Max,GrB_SECOND_FP32);

  GrB_Monoid Lor;
  GrB_Monoid_new(&Lor,GrB_LOR,(bool)false);

  GrB_Semiring Boolean;                          // Boolean semiring
  GrB_Semiring_new(&Boolean,Lor,GrB_LAND);

  // replace
  GrB_Descriptor r_desc;
  GrB_Descriptor_new(&r_desc);
  GrB_Descriptor_set(r_desc,GrB_OUTP,GrB_REPLACE);

  // replace + structural complement of mask
  GrB_Descriptor sr_desc;
  GrB_Descriptor_new(&sr_desc);
  GrB_Descriptor_set(sr_desc,GrB_MASK,GrB_SCMP);
```

```c
  GrB_Descriptor_set(sr_desc,GrB_OUTP,GrB_REPLACE);

  GrB_UnaryOp set_random;
  GrB_UnaryOp_new(&set_random,setRandom,GrB_FP32,GrB_UINT32);

  // compute the degree of each vertex.
  GrB_Vector degrees;
  GrB_Vector_new(&degrees,GrB_FP64,n);
  GrB_reduce(degrees,GrB_NULL,GrB_NULL,GrB_PLUS_FP64,A,GrB_NULL);

  // Isolated vertices are not candidates: candidates[degrees != 0] = true
  GrB_assign(candidates,degrees,GrB_NULL,true,GrB_ALL,n,GrB_NULL);

  // add all singletons to iset: iset[degree == 0] = 1
  GrB_assign(*iset,degrees,GrB_NULL,true,GrB_ALL,n,sr_desc) ;

  // Iterate while there are candidates to check.
  GrB_Index nvals;
  GrB_Vector_nvals(&nvals, candidates);
  while (nvals > 0) {
    // compute a random probability scaled by inverse of degree
    GrB_apply(prob,candidates,GrB_NULL,set_random,degrees,r_desc);

    // compute the max probability of all neighbors
    GrB_mxv(neighbor_max,candidates,GrB_NULL,maxSelect2nd,A,prob,r_desc);

    // select vertex if its probability is larger than all its active neighbors,
    // and apply a "masked no-op" to remove stored falses
    GrB_eWiseAdd(new_members,GrB_NULL,GrB_NULL,GrB_GT_FP64,prob,neighbor_max,GrB_NULL);
    GrB_apply(new_members,new_members,GrB_NULL,GrB_IDENTITY_BOOL,new_members,r_desc);

    // add new members to independent set.
    GrB_eWiseAdd(*iset,GrB_NULL,GrB_NULL,GrB_LOR,*iset,new_members,GrB_NULL);

    // remove new members from set of candidates c = c & !new
    GrB_eWiseMult(candidates,new_members,GrB_NULL,
                  GrB_LAND,candidates,candidates,sr_desc);

    GrB_Vector_nvals(&nvals, candidates);
    if (nvals == 0) { break; }                    // early exit condition

    // Neighbors of new members can also be removed from candidates
    GrB_mxv(new_neighbors,candidates,GrB_NULL,Boolean,A,new_members,GrB_NULL);
    GrB_eWiseMult(candidates,new_neighbors,GrB_NULL,
                  GrB_LAND,candidates,candidates,sr_desc);

    GrB_Vector_nvals(&nvals, candidates);
  }

  GrB_free(&neighbor_max);                        // free all objects "new'ed"
  GrB_free(&new_members);
  GrB_free(&new_neighbors);
  GrB_free(&prob);
  GrB_free(&candidates);
  GrB_free(&maxSelect2nd);
  GrB_free(&Boolean);
  GrB_free(&Max);
  GrB_free(&Lor);
  GrB_free(&sr_desc);
  GrB_free(&r_desc);
  GrB_free(&set_random);
  GrB_free(&degrees);

  return GrB_SUCCESS;
}
```

B.6 Example: counting triangles in GraphBLAS

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "GraphBLAS.h"

/*
 * Given, L, the lower triangular portion of n x n adjacency matrix A (of and
 * undirected graph), computes the number of triangles in the graph.
 */
uint64_t triangle_count(GrB_Matrix L)          // L: NxN, lower-triangular, bool
{
  GrB_Index n;
  GrB_Matrix_nrows(&n, L);                     // n = # of vertices

  GrB_Matrix C;
  GrB_Matrix_new(&C, GrB_UINT64, n, n);

  GrB_Monoid UInt64Plus;                       // integer plus monoid
  GrB_Monoid_new(&UInt64Plus,GrB_PLUS_UINT64,0ul);

  GrB_Semiring UInt64Arithmetic;               // integer arithmetic semiring
  GrB_Semiring_new(&UInt64Arithmetic,UInt64Plus,GrB_TIMES_UINT64);

  GrB_Descriptor desc_tb;                      // Descriptor for mxm
  GrB_Descriptor_new(&desc_tb);
  GrB_Descriptor_set(desc_tb,GrB_INP1,GrB_TRAN); // transpose the second matrix

  GrB_mxm(C, L, GrB_NULL, UInt64Arithmetic, L, L, desc_tb); // C<L> = L *.+ L'

  uint64_t count;
  GrB_reduce(&count, GrB_NULL, UInt64Plus, C, GrB_NULL);    // 1-norm of C

  GrB_free(&C);                       // C matrix no longer needed
  GrB_free(&UInt64Arithmetic);        // Semiring no longer needed
  GrB_free(&UInt64Plus);              // Monoid no longer needed
  GrB_free(&desc_tb);                 // descriptor no longer needed

  return count;
}
```