

1 The GraphBLAS C API Specification [†]:

2 Version 1.2.0

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

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

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

5 Copyright ©2017 Carnegie Mellon University, The Regents of the University of California, through
6 Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from the U.S.
7 Dept. of Energy), the Regents of the University of California (U.C. Davis), Intel Corporation,
8 International Business Machines Corporation, and Massachusetts Institute of Technology Lincoln
9 Laboratory.

10 Any opinions, findings and conclusions or recommendations expressed in this material are those of
11 the author(s) and do not necessarily reflect the views of the United States Department of Defense,
12 the United States Department of Energy, Carnegie Mellon University, the Regents of the University
13 of California, Intel Corporation, or the IBM Corporation.

14 NO WARRANTY. THIS MATERIAL IS FURNISHED ON AN AS-IS BASIS. THE COPYRIGHT
15 OWNERS AND/OR AUTHORS MAKE NO WARRANTIES OF ANY KIND, EITHER EX-
16 PRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WAR-
17 RANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RE-
18 SULTS OBTAINED FROM USE OF THE MATERIAL. THE COPYRIGHT OWNERS AND/OR
19 AUTHORS DO NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREE-
20 DOM FROM PATENT, TRADE MARK, OR COPYRIGHT INFRINGEMENT.

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

24 Except as otherwise noted, this material is licensed under a Creative Commons Attribution 4.0
25 license (<http://creativecommons.org/licenses/by/4.0/legalcode>), and examples are licensed under
26 the BSD License (<https://opensource.org/licenses/BSD-3-Clause>).

27 Contents

28	Acknowledgments	7
29	1 Introduction	9
30	2 Basic Concepts	11
31	2.1 Glossary	11
32	2.1.1 GraphBLAS API basic definitions	11
33	2.1.2 GraphBLAS objects and their structure	12
34	2.1.3 Algebraic structures used in the GraphBLAS	13
35	2.1.4 The execution of an application using the GraphBLAS C API	13
36	2.1.5 GraphBLAS methods: behaviors and error conditions	14
37	2.2 Notation	16
38	2.3 Algebraic and Arithmetic Foundations	17
39	2.4 GraphBLAS Opaque Objects	17
40	2.5 Domains	19
41	2.6 Operators and Associated Functions	19
42	2.7 Indices, Index Arrays, and Scalar Arrays	21
43	2.8 Execution Model	21
44	2.8.1 Execution modes	22
45	2.8.2 Thread safety	23
46	2.9 Error Model	24
47	3 Objects	27
48	3.1 Operators	27
49	3.2 Monoids	29

50	3.3 Semirings	29
51	3.4 Vectors	29
52	3.5 Matrices	30
53	3.6 Masks	31
54	3.7 Descriptors	31
55	4 Methods	35
56	4.1 Context Methods	35
57	4.1.1 init: Initialize a GraphBLAS context	35
58	4.1.2 finalize: Finalize a GraphBLAS context	36
59	4.2 Object Methods	36
60	4.2.1 Algebra Methods	37
61	4.2.1.1 Type_new: Create a new GraphBLAS (user-defined) type	37
62	4.2.1.2 UnaryOp_new: Create a new GraphBLAS unary operator	38
63	4.2.1.3 BinaryOp_new: Create a new GraphBLAS binary operator	39
64	4.2.1.4 Monoid_new: Create new GraphBLAS monoid	40
65	4.2.1.5 Semiring_new: Create new GraphBLAS semiring	41
66	4.2.2 Vector Methods	42
67	4.2.2.1 Vector_new: Create new vector	42
68	4.2.2.2 Vector_dup: Create a copy of a GraphBLAS vector	43
69	4.2.2.3 Vector_clear: Clear a vector	44
70	4.2.2.4 Vector_size: Size of a vector	45
71	4.2.2.5 Vector_nvals: Number of stored elements in a vector	46
72	4.2.2.6 Vector_build: Store elements from tuples into a vector	47
73	4.2.2.7 Vector_setElement: Set a single element in a vector	49
74	4.2.2.8 Vector_extractElement: Extract a single element from a vector.	50
75	4.2.2.9 Vector_extractTuples: Extract tuples from a vector	52
76	4.2.3 Matrix Methods	54
77	4.2.3.1 Matrix_new: Create new matrix	54
78	4.2.3.2 Matrix_dup: Create a copy of a GraphBLAS matrix	55
79	4.2.3.3 Matrix_clear: Clear a matrix	56

80	4.2.3.4	Matrix_nrows: Number of rows in a matrix	57
81	4.2.3.5	Matrix_ncols: Number of columns in a matrix	57
82	4.2.3.6	Matrix_nvals: Number of stored elements in a matrix	58
83	4.2.3.7	Matrix_build: Store elements from tuples into a matrix	59
84	4.2.3.8	Matrix_setElement: Set a single element in matrix	61
85	4.2.3.9	Matrix_extractElement: Extract a single element from a matrix . . .	62
86	4.2.3.10	Matrix_extractTuples: Extract tuples from a matrix	64
87	4.2.4	Descriptor Methods	66
88	4.2.4.1	Descriptor_new: Create new descriptor	66
89	4.2.4.2	Descriptor_set: Set content of descriptor	67
90	4.2.5	free method	68
91	4.3	GraphBLAS Operations	69
92	4.3.1	mxm: Matrix-matrix multiply	73
93	4.3.2	vxm: Vector-matrix multiply	77
94	4.3.3	mxv: Matrix-vector multiply	81
95	4.3.4	eWiseMult: Element-wise multiplication	86
96	4.3.4.1	eWiseMult: Vector variant	86
97	4.3.4.2	eWiseMult: Matrix variant	90
98	4.3.5	eWiseAdd: Element-wise addition	95
99	4.3.5.1	eWiseAdd: Vector variant	95
100	4.3.5.2	eWiseAdd: Matrix variant	100
101	4.3.6	extract: Selecting Sub-Graphs	105
102	4.3.6.1	extract: Standard vector variant	105
103	4.3.6.2	extract: Standard matrix variant	109
104	4.3.6.3	extract: Column (and row) variant	114
105	4.3.7	assign: Modifying Sub-Graphs	119
106	4.3.7.1	assign: Standard vector variant	119
107	4.3.7.2	assign: Standard matrix variant	124
108	4.3.7.3	assign: Column variant	130
109	4.3.7.4	assign: Row variant	135

110	4.3.7.5	assign: Constant vector variant	140
111	4.3.7.6	assign: Constant matrix variant	144
112	4.3.8	apply: Apply a unary function to the elements of an object	150
113	4.3.8.1	apply: Vector variant	150
114	4.3.8.2	apply: Matrix variant	154
115	4.3.9	reduce: Perform a reduction across the elements of an object	158
116	4.3.9.1	reduce: Standard matrix to vector variant	158
117	4.3.9.2	reduce: Vector-scalar variant	162
118	4.3.9.3	reduce: Matrix-scalar variant	165
119	4.3.10	transpose: Transpose rows and columns of a matrix	167
120	4.4	Sequence Termination	171
121	4.4.1	wait: Waits until pending operations complete	171
122	4.4.2	error: Get an error message regarding internal errors	172
123	5	Nonpolymorphic Interface	173
124	A	Revision History	179
125	B	Examples	181
126	B.1	Example: breadth-first search (BFS) in GraphBLAS	182
127	B.2	Example: BFS in GraphBLAS using apply	183
128	B.3	Example: betweenness centrality (BC) in GraphBLAS	184
129	B.4	Example: batched BC in GraphBLAS	186
130	B.5	Example: maximal independent set (MIS) in GraphBLAS	188
131	B.6	Example: counting triangles in GraphBLAS	190

132 Acknowledgments

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

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

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

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

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

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

154 Chapter 1

155 Introduction

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

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

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

170 The remainder of this document is organized as follows:

- 171 • Chapter 2: Basic Concepts
- 172 • Chapter 3: Objects
- 173 • Chapter 4: Methods
- 174 • Chapter 5: Nonpolymorphic Interface
- 175 • Appendix A: Revision History
- 176 • Appendix B: Examples

177 Chapter 2

178 Basic Concepts

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

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

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

190 2.1 Glossary

191 2.1.1 GraphBLAS API basic definitions

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

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

206 2.1.2 GraphBLAS objects and their structure

- 207 • *GraphBLAS object*: An instance of a data type defined by the GraphBLAS C API that
208 is opaque and manipulated only through the API. There are three groups of GraphBLAS
209 objects: *algebraic objects* (operators, monoids and semirings), *collections* (vectors, matrices
210 and masks), and descriptors. Because the object is based on an opaque datatype, an im-
211 plementation of the GraphBLAS C API has the flexibility to optimize data structures for a
212 particular platform. GraphBLAS objects are often implemented as sparse data structures,
213 meaning only the subset of the elements that have non-zero values are stored.
- 214 • *handle*: A variable that uses one of the GraphBLAS opaque data types. The value of this
215 variable holds a reference to a GraphBLAS object but not the contents of object itself. Hence,
216 assigning a value of one handle to another variable copies the reference to the GraphBLAS
217 object but not the contents of the object.
- 218 • *non-opaque datatype*: Any datatype that exposes its internal structure. This is contrasted
219 with an *opaque datatype* that hides its internal structure and can be manipulated only through
220 an API.
- 221 • *domain*: The set of valid values for the elements of a GraphBLAS object. Note that some
222 GraphBLAS objects involve functions that map values from one or more input domains onto
223 values in an output domain. These GraphBLAS objects would have multiple domains.
- 224 • *structural zero*: Any element that has a valid index (or indices) in a GraphBLAS vector or
225 matrix but is not explicitly identified in the list of elements of that vector or matrix. Also
226 known as an *implied zero*. From a mathematical perspective, a *structural zero* is treated as
227 having the value of the zero element of the relevant monoid or semiring.
- 228 • *mask*: An internal GraphBLAS object used to control how values are stored in a method's
229 output object. The mask exists only inside a method; hence, it is called an *internal opaque*
230 *object*. A mask is formed from the elements of a collection object (vector or matrix) input as
231 a mask parameter to a method. An element of the mask exists for each element that exists
232 in the input collection object when the value of that element cast to a Boolean type is `true`.
233 Masks have structure but no values. That is, while a tuple for a vector or matrix has indices
234 and values, tuples within a mask have indices but not values. Instead, we say that the tuples
235 that exist within a mask have implied values of `true` while the structural zeros of the mask
236 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

275
276
277
278
279

“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.

280
281
282
283
284
285
286
287
288

- *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.

289
290
291
292
293
294
295

- *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.

296
297
298
299
300
301
302

- *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.

303
304
305
306
307
308

- *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.

309 2.1.5 GraphBLAS methods: behaviors and error conditions

310
311
312
313

- *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.

- 314 • *thread safe routine*: A routine that performs its intended function even when executed
315 concurrently (by more than one thread).
- 316 • *shape compatible objects*: GraphBLAS objects (matrices and vectors) passed as parameters
317 to a GraphBLAS method that have the correct number of dimensions and sizes for each
318 dimension to satisfy the rules of the mathematical definition of the operation associated with
319 the method. This is also referred to as *dimension compatible*.
- 320 • *domain compatible*: Two domains for which values from one domain can be cast to values in
321 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
322 all compatible with each other, and a domain from a user-defined type is only compatible with
323 itself. If any *domain compatibility* rule above is violated, execution of GraphBLAS method
324 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.
$\odot(*)$	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.
$\otimes(S)$	Evaluates to the multiplicative binary operator of the semiring given as the argument.
$\oplus(S)$	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} .
$\tilde{\mathbf{t}}$	A temporary object created by the GraphBLAS implementation.
$\langle type \rangle$	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.

327 2.3 Algebraic and Arithmetic Foundations

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

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

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

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

358 2.4 GraphBLAS Opaque Objects

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

365 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.

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

372 An application using the GraphBLAS API will declare variables of the appropriate type for the
 373 objects it will use. Before use, the object must be initialized with the appropriate method. This
 374 is done with one of the methods that has a “`new`” suffix in its name (e.g., `GrB_Vector_new`).
 375 Alternatively, an object can be initialized by duplicating an existing object with one of the methods
 376 that has the “`dup`” suffix in its name (e.g., `GrB_Vector_dup`). When an application is finished with
 377 an object, any resources associated with that object can be released by a call to the `GrB_free`
 378 method.

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

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

391 In addition to opaque objects manipulated through handles, the GraphBLAS C API defines an
 392 additional opaque object as an internal object; that is, the object is never exposed as a variable
 393 within an application. This opaque object is the mask used to control how computed values are
 394 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	$T \rightarrow T$	$f(x) = x$, identity
GrB_UnaryOp	GrB_AINV_ T	$T \rightarrow T$	$f(x) = -x$, additive inverse
GrB_UnaryOp	GrB_MINV_ T	$T \rightarrow T$	$f(x) = \frac{1}{x}$, multiplicative inverse
GrB_UnaryOp	GrB_LNOT	bool \rightarrow bool	$f(x) = \neg x$, logical inverse
GrB_BinaryOp	GrB_LOR	bool \times bool \rightarrow bool	$f(x, y) = x \vee y$, logical OR
GrB_BinaryOp	GrB_LAND	bool \times bool \rightarrow bool	$f(x, y) = x \wedge y$, logical AND
GrB_BinaryOp	GrB_LXOR	bool \times bool \rightarrow bool	$f(x, y) = x \oplus y$, logical XOR
GrB_BinaryOp	GrB_EQ_ T	$T \times T \rightarrow$ bool	$f(x, y) = (x == y)$ equal
GrB_BinaryOp	GrB_NE_ T	$T \times T \rightarrow$ bool	$f(x, y) = (x \neq y)$ not equal
GrB_BinaryOp	GrB_GT_ T	$T \times T \rightarrow$ bool	$f(x, y) = (x > y)$ greater than
GrB_BinaryOp	GrB_LT_ T	$T \times T \rightarrow$ bool	$f(x, y) = (x < y)$ less than
GrB_BinaryOp	GrB_GE_ T	$T \times T \rightarrow$ bool	$f(x, y) = (x \geq y)$ greater than or equal
GrB_BinaryOp	GrB_LE_ T	$T \times T \rightarrow$ bool	$f(x, y) = (x \leq y)$ less than or equal
GrB_BinaryOp	GrB_FIRST_ T	$T \times T \rightarrow T$	$f(x, y) = x$, first argument
GrB_BinaryOp	GrB_SECOND_ T	$T \times T \rightarrow T$	$f(x, y) = y$, second argument
GrB_BinaryOp	GrB_MIN_ T	$T \times T \rightarrow T$	$f(x, y) = (x < y) ? x : y$, minimum
GrB_BinaryOp	GrB_MAX_ T	$T \times T \rightarrow T$	$f(x, y) = (x > y) ? x : y$, maximum
GrB_BinaryOp	GrB_PLUS_ T	$T \times T \rightarrow T$	$f(x, y) = x + y$, addition
GrB_BinaryOp	GrB_MINUS_ T	$T \times T \rightarrow T$	$f(x, y) = x - y$, subtraction
GrB_BinaryOp	GrB_TIMES_ T	$T \times T \rightarrow T$	$f(x, y) = xy$, multiplication
GrB_BinaryOp	GrB_DIV_ T	$T \times T \rightarrow T$	$f(x, y) = \frac{x}{y}$, division

414 2.7 Indices, Index Arrays, and Scalar Arrays

415 In order to interface with third-party software (i.e., software other than an implementation of
416 the GraphBLAS), operations such as `GrB_Matrix_build` (§ 4.2.3.7) and `GrB_Matrix_extractTuples`
417 (§ 4.2.3.10) must specify how the data should be laid out in non-opaque data structures. To this
418 end we explicitly define the types for indices and the arrays used by these operations.

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

```
420     typedef uint64_t GrB_Index;
```

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

432 2.8 Execution Model

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

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

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

449 The GraphBLAS objects are fully defined at any point in a sequence by the methods in the sequence
450 as long as there are no execution errors. In particular, as soon as a GraphBLAS method call returns,

451 its output can be used in the next GraphBLAS method call. However, individual operations in a
 452 sequence may not be *complete*. We say that an operation is complete when all the computations
 453 in the operation have finished and all the values of its output object have been produced and
 454 committed to the address space of the program.

455 The opaqueness of GraphBLAS objects allows execution to proceed from one method to the next
 456 even when operations are not complete. Processing of nonopaque objects is never deferred in
 457 GraphBLAS. That is, methods that consume nonopaque objects (e.g., `GrB_Matrix_build`, § 4.2.3.7())
 458 and methods that produce nonopaque objects (e.g., `GrB_Matrix_extractTuples()`, § 4.2.3.10) always
 459 finish consuming or producing those nonopaque objects before returning. Furthermore, methods
 460 that extract values from opaque GraphBLAS objects into nonopaque user objects (see Table 2.4)
 461 always force completion of all pending computations on the corresponding GraphBLAS source
 462 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
<code>GrB_Vector_nvals</code>	4.2.2.5
<code>GrB_Vector_extractElement</code>	4.2.2.8
<code>GrB_Vector_extractTuples</code>	4.2.2.9
<code>GrB_Matrix_nvals</code>	4.2.3.6
<code>GrB_Matrix_extractElement</code>	4.2.3.9
<code>GrB_Matrix_extractTuples</code>	4.2.3.10
<code>GrB_reduce</code> (vector-scalar variant)	4.3.9.2
<code>GrB_reduce</code> (matrix-scalar variant)	4.3.9.3

463 2.8.1 Execution modes

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

- 466 • *blocking*: In blocking mode, each method completes the GraphBLAS operation defined by
 467 the method before proceeding to the next statement in program order. Output GraphBLAS
 468 objects defined by a method are fully produced and stored in memory (i.e., they are *mate-*
 469 *rialized*). In other words, it is as if each method call is its own sequence. Even mechanisms
 470 that break the opaqueness of the GraphBLAS objects (e.g., performance monitors, debuggers,
 471 memory dumps) will observe the operation as complete.
- 472 • *nonblocking*: In nonblocking mode, each method may return once the input arguments have
 473 been inspected and verified to define a well formed GraphBLAS operation. (That is, there
 474 are no API errors. See § 2.9.) The GraphBLAS operation may not have completed, but the
 475 output object is ready to be used by the next GraphBLAS method call. Completion of *all*
 476 operations in a sequence, including any that may generate execution errors, is guaranteed
 477 once the sequence terminates. Sequence termination is accomplished by a call to `GrB_wait()`.

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

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

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

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

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

- 501 • `GrB_BLOCKING` Specifies the blocking mode context.
- 502 • `GrB_NONBLOCKING` Specifies the blocking mode context.

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

506 2.8.2 Thread safety

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

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

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

522 2.9 Error Model

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

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

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

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

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

554 If a GraphBLAS method, executing in nonblocking mode, returns with any execution error from
555 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.

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

560 After a call to any GraphBLAS method, the program can retrieve additional error information
561 (beyond the error code returned by the method) through a call to the function `GrB_error()`. The
562 signature of that function is shown in Figure 2.1. The function returns a pointer to a NULL-
563 terminated string, and the contents of that string are implementation dependent. In particular, a
564 null string (not a NULL pointer) is always a valid error string. The pointer is valid until the next
565 call to any GraphBLAS method by the same thread. `GrB_error()` is a thread-safe function, in the
566 sense that multiple threads can call it simultaneously and each will get its own error string back,
567 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 <code>new</code> 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 $\odot(F_b) = \odot$. 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

589 3.2 Monoids

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

594 Let $F = \langle D, D, D, \odot \rangle$ be an associative GraphBLAS binary operator with identity element $0 \in D$.
595 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
596 a *commutative monoid*. If a monoid M is created using an operator \odot that is not associative, the
597 outcome of GraphBLAS operations using such a monoid is undefined.

598 3.3 Semirings

599 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
600 domains D_{out} , D_{in_1} , and D_{in_2} ; an *associative*² and commutative additive operation $\oplus : D_{out} \times$
601 $D_{out} \rightarrow D_{out}$; a multiplicative operation $\otimes : D_{in_1} \times D_{in_2} \rightarrow D_{out}$; and an identity element $0 \in$
602 D_{out} . For a given GraphBLAS semiring $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$ we define $\mathbf{D}_{in_1}(S) = D_{in_1}$,
603 $\mathbf{D}_{in_2}(S) = D_{in_2}$, $\mathbf{D}_{out}(S) = D_{out}$, $\oplus(S) = \oplus$, $\otimes(S) = \otimes$, and $\mathbf{0}(S) = 0$.

604 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,
605 then $S = \langle A, F \rangle = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$ is a semiring.

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

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

612 3.4 Vectors

613 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)
614 where $0 \leq i < N$ and $v_i \in D$. A particular value of i can appear at most once in \mathbf{v} . We define
615 $\mathbf{size}(\mathbf{v}) = N$ 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
616 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} ,
617 $\mathbf{v}(i)$ is a reference to v_i if $(i, v_i) \in \mathbf{L}(\mathbf{v})$ and is undefined otherwise.

¹It 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.

²It 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.

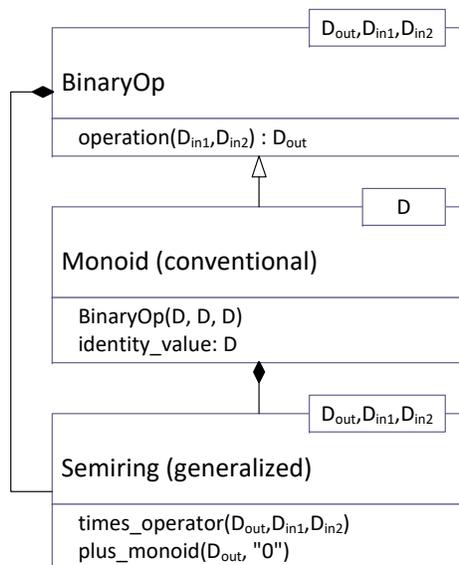


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

618 3.5 Matrices

619 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
 620 number of columns $N > 0$, and a set of tuples (i, j, A_{ij}) where $0 \leq i < M$, $0 \leq j < N$, and
 621 $A_{ij} \in D$. A particular pair of values i, j can appear at most once in \mathbf{A} . We define $\mathbf{ncols}(\mathbf{A}) = N$,
 622 $\mathbf{nrows}(\mathbf{A}) = M$, 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
 623 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
 624 are the sets of nonempty rows and columns of \mathbf{A} , respectively.) The *structure* of matrix \mathbf{A} is the
 625 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
 626 A_{ij} if $(i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})$ and is undefined otherwise.

627 If \mathbf{A} is a matrix and $0 \leq j < N$, then $\mathbf{A}(:, j) = \langle D, M, \{(i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\} \rangle$ is a
 628 vector called the j -th *column* of \mathbf{A} . Correspondingly, if \mathbf{A} is a matrix and $0 \leq i < M$, then
 629 $\mathbf{A}(i, :) = \langle D, N, \{(j, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\} \rangle$ is a vector called the i -th *row* of \mathbf{A} .

630 Given a matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$, its *transpose* is another matrix $\mathbf{A}^T = \langle D, N, M, \{(j, i, A_{ij}) : \}$
 631 $(i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\} \rangle$.

632 3.6 Masks

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

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

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

647 A one-dimensional mask $\mathbf{m} = \langle N, \{i\} \rangle$ is defined by its number of elements $N > 0$, and a set
648 $\mathbf{ind}(\mathbf{m})$ of indices $\{i\}$ where $0 \leq i < N$. A particular value of i can appear at most once in \mathbf{m} . We
649 define $\mathbf{size}(\mathbf{m}) = N$. The set $\mathbf{ind}(\mathbf{m})$ is called the *structure* of mask \mathbf{m} .

650 A two-dimensional mask $\mathbf{M} = \langle M, N, \{(i, j)\} \rangle$ is defined by its number of rows $M > 0$, its number
651 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
652 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
653 also define the sets $\mathbf{indrow}(\mathbf{M}) = \{i : \exists(i, j) \in \mathbf{ind}(\mathbf{M})\}$ and $\mathbf{indcol}(\mathbf{M}) = \{j : \exists(i, j) \in \mathbf{ind}(\mathbf{M})\}$.
654 These are the sets of nonempty rows and columns of \mathbf{M} , respectively. The set $\mathbf{ind}(\mathbf{M})$ is called the
655 *structure* of mask \mathbf{M} .

656 One common operation on masks is the *structural complement*. For a one-dimensional mask \mathbf{m}
657 this is denoted as $\neg\mathbf{m}$. For a two-dimensional masks, this is denoted as $\neg\mathbf{M}$. The structure of the
658 complement of a one-dimensional mask \mathbf{m} is defined as $\mathbf{ind}(\neg\mathbf{m}) = \{i : 0 \leq i < N, i \notin \mathbf{ind}(\mathbf{m})\}$.
659 It is the set of all possible indices that do not appear in \mathbf{m} . The structure of the complement of
660 a two-dimensional mask \mathbf{M} is defined as the set $\mathbf{ind}(\neg\mathbf{M}) = \{(i, j) : 0 \leq i < M, 0 \leq j < N,$
661 $(i, j) \notin \mathbf{ind}(\mathbf{M})\}$. It is the set of all possible indices that do not appear in \mathbf{M} .

662 3.7 Descriptors

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

667 The descriptor is a lightweight object. It is composed of (field, value) pairs where the *field* selects one
668 of the GraphBLAS objects from the argument list of a method and the *value* defines the indicated
669 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
<code>GrB_Descriptor</code>	Type of a GraphBLAS descriptor object.
<code>GrB_Desc_Field</code>	Type of a descriptor field.
<code>GrB_Desc_Value</code>	Type of a descriptor field’s value.

(b) Descriptor field names of type `GrB_Desc_Field`.

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

(c) Descriptor field values of type `GrB_Desc_Value`.

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

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

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

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

685 defined as follows:

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

689 Chapter 4

690 Methods

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

693 4.1 Context Methods

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

697 4.1.1 `init`: Initialize a GraphBLAS context

698 Creates and initializes a GraphBLAS C API context.

699 C Syntax

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

701 Parameters

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

703 Return Values

704 `GrB_SUCCESS` operation completed successfully.

705 `GrB_PANIC` unknown internal error.

706 `GrB_INVALID_VALUE` invalid mode specified, or method called multiple times.

707 **Description**

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

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

719 An application can only create one context per execution instance.

720 **4.1.2 finalize: Finalize a GraphBLAS context**

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

722 **C Syntax**

```
723         GrB_Info GrB_finalize();
```

724 **Return Values**

725 `GrB_SUCCESS` operation completed successfully.

726 `GrB_PANIC` unknown internal error.

727 **Description**

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

731 **4.2 Object Methods**

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

734 4.2.1 Algebra Methods

735 4.2.1.1 Type_new: Create a new GraphBLAS (user-defined) type

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

738 C Syntax

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

741 Parameters

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

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

745 Return Values

746 GrB_SUCCESS operation completed successfully.

747 GrB_PANIC unknown internal error.

748 GrB_OUT_OF_MEMORY not enough memory available for operation.

749 GrB_NULL_POINTER utype pointer is NULL.

750 Description

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

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

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

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

760 4.2.1.2 UnaryOp_new: Create a new GraphBLAS unary operator

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

763 C Syntax

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

768 Parameters

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

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

```
774         void func(void *out, const void *in);  
775
```

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

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

782 Return Values

783 GrB_SUCCESS operation completed successfully.

784 GrB_PANIC unknown internal error.

785 GrB_OUT_OF_MEMORY not enough memory available for operation.

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

788 GrB_NULL_POINTER unary_op or unary_func pointers are NULL.

789 **Description**

790 Creates a new GraphBLAS unary operator $f_u = \langle \mathbf{D}(d_out), \mathbf{D}(d_in), unary_func \rangle$ and returns a
791 handle to it in `unary_op`.

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

794 **4.2.1.3 BinaryOp_new: Create a new GraphBLAS binary operator**

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

797 **C Syntax**

```
798         GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
799                                 void          (*binary_func)(void*,  
800                                                         const void*,  
801                                                         const void*),  
802                                 GrB_Type      d_out,  
803                                 GrB_Type      d_in1,  
804                                 GrB_Type      d_in2);
```

805 **Parameters**

806 `binary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
807 binary operator object.

808 `binary_func` (IN) A pointer to a user-defined function that takes two input parameters of types
809 `d_in1` and `d_in2` and returns a value of type `d_out`, all passed as void pointers.
810 Specifically the signature of the function is expected to be of the form:

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

813 `d_out` (IN) The `GrB_Type` of the return value of the binary operator being created. Should
814 be one of the predefined GraphBLAS types in Table 2.2, or a user-defined Graph-
815 BLAS type.

816 `d_in1` (IN) The `GrB_Type` of the left hand argument of the binary operator being created.
817 Should be one of the predefined GraphBLAS types in Table 2.2, or a user-defined
818 GraphBLAS type.

819 `d_in2` (IN) The `GrB_Type` of the right hand argument of the binary operator being created.
820 Should be one of the predefined GraphBLAS types in Table 2.2, or a user-defined
821 GraphBLAS type.

822 **Return Values**

823 GrB_SUCCESS operation completed successfully.

824 GrB_PANIC unknown internal error.

825 GrB_OUT_OF_MEMORY not enough memory available for operation.

826 GrB_UNINITIALIZED_OBJECT the GrB_Type (for user-defined types) has not been initialized by a
827 call to GrB_Type_new.

828 GrB_NULL_POINTER binary_op or binary_func pointer is NULL.

829 **Description**

830 Creates a new GraphBLAS binary operator $f_b = \langle \mathbf{D}(d_out), \mathbf{D}(d_in1), \mathbf{D}(d_in2), \text{binary_func} \rangle$ and
831 returns a handle to it in binary_op.

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

834 **4.2.1.4 Monoid_new: Create new GraphBLAS monoid**

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

836 **C Syntax**

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

840 **Parameters**

841 monoid (INOUT) On successful return, contains a handle to the newly created GraphBLAS
842 monoid object.

843 binary_op (IN) An existing GraphBLAS associative binary operator whose input and output
844 types are the same.

845 identity (IN) The value of the identity element of the monoid. Must be the same type as
846 the type used by the binary_op operator.

847 **Return Values**

848 GrB_SUCCESS operation completed successfully.

849 GrB_PANIC unknown internal error.

850 GrB_OUT_OF_MEMORY not enough memory available for operation.

851 GrB_UNINITIALIZED_OBJECT the GrB_BinaryOp has not been initialized by a call to GrB_BinaryOp_new.

852 GrB_NULL_POINTER monoid pointer is NULL.

853 GrB_DOMAIN_MISMATCH all three argument types of the binary operator and the type of the
854 identity value are not the same.

855 **Description**

856 Creates a new monoid $M = \langle \mathbf{D}(\text{binary_op}), \text{binary_op}, \text{identity} \rangle$ and returns a handle to it in monoid.
857 If `binary_op` is not associative, the results of GraphBLAS operations that require associativity of
858 this monoid will be undefined.

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

861 **4.2.1.5 Semiring_new: Create new GraphBLAS semiring**

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

863 **C Syntax**

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

867 **Parameters**

868 semiring (INOUT) On successful return, contains a handle to the newly created GraphBLAS
869 semiring.

870 add_op (IN) An existing GraphBLAS commutative monoid that specifies the addition op-
871 erator and its identity.

872 mul_op (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-
873 plication operator. In addition, `mul_op`'s output domain, $\mathbf{D}_{out}(\text{mul_op})$, must be
874 the same as the `add_op`'s domain $\mathbf{D}(\text{add_op})$.

875 Return Values

876 `GrB_SUCCESS` operation completed successfully.

877 `GrB_PANIC` unknown internal error.

878 `GrB_OUT_OF_MEMORY` not enough memory available for this method to complete.

879 `GrB_UNINITIALIZED_OBJECT` the `add_op` object has not been initialized with a call to `GrB_Monoid_new`
880 or the `mul_op` object has not been initialized by a call to
881 `GrB_BinaryOp_new`.

882 `GrB_NULL_POINTER` semiring pointer is `NULL`.

883 `GrB_DOMAIN_MISMATCH` the output domain of `mul_op` does not match the domain of the
884 `add_op` monoid.

885 Description

886 Creates a new semiring $S = \langle \mathbf{D}_{out}(\text{mul_op}), \mathbf{D}_{in_1}(\text{mul_op}), \mathbf{D}_{in_2}(\text{mul_op}), \text{add_op}, \text{mul_op}, \mathbf{0}(\text{add_op}) \rangle$
887 and returns a handle to it in `semiring`. Note that $\mathbf{D}_{out}(\text{mul_op})$ must be the same as $\mathbf{D}(\text{add_op})$.

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

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

891 4.2.2 Vector Methods

892 4.2.2.1 `Vector_new`: Create new vector

893 Creates a new vector with specified domain and size.

894 C Syntax

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

898 Parameters

899 `v` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
900 vector.

901 `d` (IN) The type corresponding to the domain of the vector being created. Can be
902 one of the predefined GraphBLAS types in Table 2.2, or an existing user-defined
903 GraphBLAS type.

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

905 **Return Values**

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

910 `GrB_PANIC` Unknown internal error.

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

915 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

916 `GrB_UNINITIALIZED_OBJECT` The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`
917 (needed for user-defined types).

918 `GrB_NULL_POINTER` The `v` pointer is `NULL`.

919 `GrB_INVALID_VALUE` `nsize` is zero.

920 **Description**

921 Creates a new vector `v` of domain `D(d)`, size `nsize`, and empty `L(v)`. The method returns a handle
922 to the new vector in `v`.

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

925 **4.2.2.2 Vector_dup: Create a copy of a GraphBLAS vector**

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

927 **C Syntax**

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

930 **Parameters**

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

933 u (IN) The GraphBLAS vector to be duplicated.

934 Return Values

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

939 `GrB_PANIC` Unknown internal error.

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

944 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

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

947 `GrB_NULL_POINTER` The w pointer is `NULL`.

948 Description

949 Creates a new vector w of domain $D(u)$, size $size(u)$, and contents $L(u)$. The method returns a
950 handle to the new vector in w .

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

953 4.2.2.3 `Vector_clear`: Clear a vector

954 Removes all the elements (tuples) from a vector.

955 C Syntax

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

957 Parameters

958 v (INOUT) An existing GraphBLAS vector to clear.

959 Return Values

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

964 GrB_PANIC Unknown internal error.

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

969 GrB_OUT_OF_MEMORY Not enough memory available for operation.

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

972 Description

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

975 4.2.2.4 `Vector_size`: Size of a vector

976 Retrieve the size of a vector.

977 C Syntax

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

980 Parameters

981 $nsize$ (OUT) On successful return, is set to the size of the vector.

982 v (IN) An existing GraphBLAS vector being queried.

983 Return Values

984 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
985 cessfully and the value of $nsize$ has been set.

986 GrB_PANIC Unknown internal error.

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

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

993 GrB_NULL_POINTER nsize pointer is NULL.

994 **Description**

995 Return `size(v)` in `nsize`.

996 **4.2.2.5 Vector_nvals: Number of stored elements in a vector**

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

998 **C Syntax**

```
999           GrB_Info GrB_Vector_nvals(GrB_Index            *nvals,
1000                                    const GrB_Vector   v);
```

1001 **Parameters**

1002 nvals (OUT) On successful return, this is set to the number of stored elements (tuples)
1003 in the vector.

1004 v (IN) An existing GraphBLAS vector being queried.

1005 **Return Values**

1006 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1007 cessfully and the value of `nvals` has been set.

1008 GrB_PANIC Unknown internal error.

1009 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1010 GraphBLAS objects (input or output) is in an invalid state caused
1011 by a previous execution error. Call GrB_error() to access any error
1012 messages generated by the implementation.

1013 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1014 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v , has not been initialized by a call to
1015 Vector_new or Vector_dup.

1016 GrB_NULL_POINTER The nvals pointer is NULL.

1017 Description

1018 Return $nvals(v)$ in nvals. This is the number of stored elements in vector v , which is the size of
1019 $L(v)$ (see Section 3.4).

1020 4.2.2.6 Vector_build: Store elements from tuples into a vector

1021 C Syntax

```
1022     GrB_Info GrB_Vector_build(GrB_Vector      w,  
1023                             const GrB_Index  *indices,  
1024                             const <type>    *values,  
1025                             GrB_Index       n,  
1026                             const GrB_BinaryOp dup);
```

1027 Parameters

1028 w (INOUT) An existing Vector object to store the result.

1029 indices (IN) Pointer to an array of indices.

1030 values (IN) Pointer to an array of scalars of a type that is compatible with the domain of
1031 vector w .

1032 n (IN) The number of entries contained in each array (the same for indices and values).

1033 dup (IN) An associative and commutative binary operator to apply when duplicate
1034 values for the same location are present in the input arrays. All three domains of
1035 dup must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$.

1036 Return Values

1037 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1038 blocking mode, this indicates that the API checks for the input
1039 arguments passed successfully. Either way, output vector w is ready
1040 to be used in the next method of the sequence.

1041 GrB_PANIC Unknown internal error.

1042 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 1043 GraphBLAS objects (input or output) is in an invalid state caused
 1044 by a previous execution error. Call `GrB_error()` to access any error
 1045 messages generated by the implementation.

1046 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1047 GrB_UNINITIALIZED_OBJECT Either `w` has not been initialized by a call to `GrB_Vector_new` or
 1048 by `GrB_Vector_dup`, or `dup` has not been initialized by a call to `GrB_BinaryOp_new`.
 1049

1050 GrB_NULL_POINTER `indices` or `values` pointer is `NULL`.

1051 GrB_INDEX_OUT_OF_BOUNDS A value in `indices` is outside the allowed range for `w`.

1052 GrB_DOMAIN_MISMATCH Either the domains of the GraphBLAS binary operator `dup` are not
 1053 all the same, or the domains of `values` and `w` are incompatible with
 1054 each other or D_{dup} .

1055 GrB_OUTPUT_NOT_EMPTY Output vector `w` already contains valid tuples (elements). In other
 1056 words, `GrB_Vector_nvals(C)` returns a positive value.

1057 Description

1058 An internal vector $\tilde{\mathbf{w}} = \langle D_{dup}, \mathbf{size}(w), \emptyset \rangle$ is created, which only differs from `w` in its domain.

1059 Each tuple $\{\text{indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the output in the form of

$$\tilde{\mathbf{w}}(\text{indices}[k]) = (D_{dup}) \text{values}[k].$$

1060 If multiple values for the same location are present in the input arrays, the `dup` binary operand is
 1061 used to reduce them before assignment into $\tilde{\mathbf{w}}$ as follows:

$$1062 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1063 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{w}}$ is copied into `w` via typecasting its
 1064 values to $\mathbf{D}(w)$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

1065 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1066 It is an error to call this function on an output object with existing elements. In other words,
 1067 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1068 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices` or
 1069 `values`.

1070 **4.2.2.7 Vector_setElement: Set a single element in a vector**

1071 Set one element of a vector to a given value.

1072 **C Syntax**

```
1073         GrB_Info GrB_Vector_setElement(GrB_Vector  w,  
1074                                     <type>      val,  
1075                                     GrB_Index   index);
```

1076 **Parameters**

1077 w (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1078 val (IN) Scalar value to assign. The type must be compatible with the domain of w.

1079 index (IN) The location of the element to be assigned.

1080 **Return Values**

1081 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1082 blocking mode, this indicates that the compatibility tests on in-
1083 dex/dimensions and domains for the input arguments passed suc-
1084 cessfully. Either way, the output vector w is ready to be used in
1085 the next method of the sequence.

1086 GrB_PANIC Unknown internal error.

1087 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1088 GraphBLAS objects (input or output) is in an invalid state caused
1089 by a previous execution error. Call GrB_error() to access any error
1090 messages generated by the implementation.

1091 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1092 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, w, has not been initialized by a call to
1093 Vector_new or Vector_dup.

1094 GrB_INVALID_INDEX index specifies a location that is outside the dimensions of w.

1095 GrB_DOMAIN_MISMATCH The domains of w and val are incompatible.

1096 **Description**

1097 First, the scalar and output vector are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ must be
1098 compatible with $\mathbf{D}(w)$. Two domains are compatible with each other if values from one domain can

1099 be cast to values in the other domain as per the rules of the C language. In particular, domains from
1100 Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible
1101 with itself. If any compatibility rule above is violated, execution of `GrB_Vector_setElement` ends and
1102 the domain mismatch error listed above is returned.

1103 Then, the `index` parameter is checked for a valid value where the following condition must hold:

$$1104 \quad 0 \leq \text{index} < \text{size}(w)$$

1105 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the invalid index error
1106 listed above is returned.

1107 We are now ready to carry out the assignment `val`; that is:

$$1108 \quad w(\text{index}) = \text{val}$$

1109 If a value existed at this location in `w`, it will be overwritten; otherwise, a new value is stored in
1110 `w`.

1111 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1112 of `w` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1113 return value `GrB_SUCCESS` and the new content of vector `w` is as defined above but may not be
1114 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1115 **4.2.2.8 Vector_extractElement: Extract a single element from a vector.**

1116 Extract one element of a vector into a scalar.

1117 **C Syntax**

```
1118     GrB_Info GrB_Vector_extractElement(<type>          *val,  
1119                                     const GrB_Vector u,  
1120                                     GrB_Index       index);
```

1121 **Parameters**

1122 `val` (INOUT) Pointer to a scalar of type that is compatible with the domain of vector
1123 `w`. On successful return, this scalar holds the result of the operation. Any previous
1124 value in `val` is overwritten.

1125 `u` (IN) The GraphBLAS vector from which an element is extracted.

1126 `index` (IN) The location in `u` to extract.

1127 Return Values

1128 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1129 cessfully. This indicates that the compatibility tests on dimensions
1130 and domains for the input arguments passed successfully, and the
1131 output scalar, `val`, has been computed and is ready to be used in
1132 the next method of the sequence.

1133 GrB_PANIC Unknown internal error.

1134 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1135 GraphBLAS objects (input or output) is in an invalid state caused
1136 by a previous execution error. Call `GrB_error()` to access any error
1137 messages generated by the implementation.

1138 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1139 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `u`, has not been initialized by a call to
1140 `Vector_new` or `Vector_dup`.

1141 GrB_NULL_POINTER `val` pointer is NULL.

1142 GrB_NO_VALUE There is no stored value at specified location.

1143 GrB_INVALID_INDEX `index` specifies a location that is outside the dimensions of `w`.

1144 GrB_DOMAIN_MISMATCH The domains of the vector or scalar are incompatible.

1145 Description

1146 First, the scalar and input vector are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ must be
1147 compatible with $\mathbf{D}(u)$. Two domains are compatible with each other if values from one domain can
1148 be cast to values in the other domain as per the rules of the C language. In particular, domains from
1149 Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible
1150 with itself. If any compatibility rule above is violated, execution of `GrB_Vector_extractElement` ends
1151 and the domain mismatch error listed above is returned.

1152 Then, the `index` parameter is checked for a valid value where the following condition must hold:

$$1153 \quad 0 \leq \text{index} < \text{size}(u)$$

1154 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the invalid index error
1155 listed above is returned.

1156 We are now ready to carry out the extract into the output argument, `val`; that is:

$$1157 \quad \text{val} = u(\text{index})$$

1158 where the following condition must be true:

$$1159 \quad \text{index} \in \mathbf{ind}(u)$$

1160 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the "no value" error
1161 listed above is returned.

1162 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1163 `GrB_SUCCESS`, the new contents of `val` are as defined above. In other words, the method does not
1164 return until any operations required to fully compute the GraphBLAS vector `u` have completed.

1165 In `GrB_NONBLOCKING` mode, if the return value is not `GrB_SUCCESS`, an error in a method
1166 occurring earlier in the sequence may have occurred that prevents completion of the GraphBLAS
1167 vector `u`. The `GrB_error()` method should be called for additional information about these errors.

1168 4.2.2.9 `Vector_extractTuples`: Extract tuples from a vector

1169 Extract the contents of a GraphBLAS vector into non-opaque data structures.

1170 C Syntax

```
1171     GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,  
1172                                     <type>        *values,  
1173                                     GrB_Index      *n,  
1174                                     const GrB_Vector v);  
1175
```

1176 `indices` (OUT) Pointer to an array of indices that is large enough to hold all of the stored
1177 values' indices.

1178 `values` (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
1179 the stored values whose type is compatible with $\mathbf{D}(\mathbf{v})$.

1180 `n` (INOUT) Pointer to a value indicating (on input) the number of elements the
1181 `values` and `indices` arrays can hold. Upon return, it will contain the number of
1182 values written to the arrays.

1183 `v` (IN) An existing GraphBLAS vector.

1184 Return Values

1185 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
1186 cessfully. This indicates that the compatibility tests on the input
1187 argument passed successfully, and the output arrays, `indices` and
1188 `values`, have been computed.

1189 `GrB_PANIC` Unknown internal error.

1190 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1191 GraphBLAS objects (input or output) is in an invalid state caused

1192 by a previous execution error. Call `GrB_error()` to access any error
1193 messages generated by the implementation.

1194 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1195 `GrB_INSUFFICIENT_SPACE` Not enough space in indices and values (as indicated by the `n` pa-
1196 rameter) to hold all of the tuples that will be extracted.

1197 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS vector, `v`, has not been initialized by a call to
1198 `Vector_new` or `Vector_dup`.

1199 `GrB_NULL_POINTER` indices, values, or `n` pointer is `NULL`.

1200 `GrB_DOMAIN_MISMATCH` The domains of the `v` vector or values array are incompatible with
1201 one another.

1202 **Description**

1203 This method will extract all the tuples from the GraphBLAS vector `v`. The values associated
1204 with those tuples are placed in the `values` array and the indices are placed in the `indices` array.
1205 Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least
1206 `GrB_Vector_nvals(v)` elements before calling this function.

1207 Upon return of this function, `n` will be set to the number of values (and indices) copied. Also, the
1208 entries of `indices` are unique, but not necessarily sorted. Each tuple (i, v_i) in `v` is unzipped and
1209 copied into a distinct k th location in output vectors:

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

1210 where $0 \leq k < \text{GrB_Vector_nvals}(v)$. No gaps in output vectors are allowed; that is, if `indices[k]` and
1211 `values[k]` exist upon return, so does `indices[j]` and `values[j]` for all j such that $0 \leq j < k$.

1212 Note that if the value in `n` on input is less than the number of values contained in the vector `v`,
1213 then a `GrB_INSUFFICIENT_SPACE` error is returned because it is undefined which subset of values
1214 would be extracted otherwise.

1215 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1216 `GrB_SUCCESS`, the new contents of the arrays `indices` and `values` are as defined above. In other
1217 words, the method does not return until any operations required to fully compute the GraphBLAS
1218 vector `v` have completed.

1219 In `GrB_NONBLOCKING` mode, if the return value is not `GrB_SUCCESS`, an error in a method
1220 occurring earlier in the sequence may have occurred that prevents completion of the GraphBLAS
1221 vector `v`. The `GrB_error()` method should be called for additional information about these errors.

1222 4.2.3 Matrix Methods

1223 4.2.3.1 Matrix_new: Create new matrix

1224 Creates a new matrix with specified domain and dimensions.

1225 C Syntax

```
1226     GrB_Info GrB_Matrix_new(GrB_Matrix *A,  
1227                             GrB_Type    d,  
1228                             GrB_Index   nrows,  
1229                             GrB_Index   ncols);
```

1230 Parameters

1231 A (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1232 matrix.

1233 d (IN) The type corresponding to the domain of the matrix being created. Can be
1234 one of the predefined GraphBLAS types in Table 2.2, or an existing user-defined
1235 GraphBLAS type.

1236 nrows (IN) The number of rows of the matrix being created.

1237 ncols (IN) The number of columns of the matrix being created.

1238 Return Values

1239 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1240 blocking mode, this indicates that the API checks for the input arg-
1241 uments passed successfully. Either way, output matrix A is ready
1242 to be used in the next method of the sequence.

1243 GrB_PANIC Unknown internal error.

1244 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1245 GraphBLAS objects (input or output) is in an invalid state caused
1246 by a previous execution error. Call GrB_error() to access any error
1247 messages generated by the implementation.

1248 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1249 GrB_UNINITIALIZED_OBJECT The GrB_Type object has not been initialized by a call to GrB_Type_new
1250 (needed for user-defined types).

1251 GrB_NULL_POINTER The A pointer is NULL.

1252 GrB_INVALID_VALUE nrows or ncols is zero.

1253 **Description**

1254 Creates a new matrix **A** of domain **D**(**d**), size **nrows** × **ncols**, and empty **L(A)**. The method returns
1255 a handle to the new matrix in **A**.

1256 It is not an error to call this method more than once on the same variable; however, the handle to
1257 the previously created object will be overwritten.

1258 **4.2.3.2 Matrix_dup: Create a copy of a GraphBLAS matrix**

1259 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

1260 **C Syntax**

```
1261         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
1262                               const GrB_Matrix A);
```

1263 **Parameters**

1264 **C** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1265 matrix.

1266 **A** (IN) The GraphBLAS matrix to be duplicated.

1267 **Return Values**

1268 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1269 blocking mode, this indicates that the API checks for the input
1270 arguments passed successfully. Either way, output matrix **C** is ready
1271 to be used in the next method of the sequence.

1272 **GrB_PANIC** Unknown internal error.

1273 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1274 GraphBLAS objects (input or output) is in an invalid state caused
1275 by a previous execution error. Call **GrB_error()** to access any error
1276 messages generated by the implementation.

1277 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1278 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
1279 **Matrix_new** or **Matrix_dup**.

1280 **GrB_NULL_POINTER** The **C** pointer is **NULL**.

1281 **Description**

1282 Creates a new matrix **C** of domain **D(A)**, size **nrows(A) × ncols(A)**, and contents **L(A)**. It returns
1283 a handle to it in **C**.

1284 It is not an error to call this method more than once on the same variable; however, the handle to
1285 the previously created object will be overwritten.

1286 **4.2.3.3 Matrix_clear: Clear a matrix**

1287 Removes all elements (tuples) from a matrix.

1288 **C Syntax**

1289 `GrB_Info GrB_Matrix_clear(GrB_Matrix A);`

1290 **Parameters**

1291 **A (IN)** An existing GraphBLAS matrix to clear.

1292 **Return Values**

1293 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
1294 blocking mode, this indicates that the API checks for the input argu-
1295 ments passed successfully. Either way, output matrix **A** is ready
1296 to be used in the next method of the sequence.

1297 **GrB_PANIC** Unknown internal error.

1298 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1299 GraphBLAS objects (input or output) is in an invalid state caused
1300 by a previous execution error. Call `GrB_error()` to access any error
1301 messages generated by the implementation.

1302 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1303 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, ***A**, has not been initialized by a call to
1304 `Matrix_new` or `Matrix_dup`.

1305 **Description**

1306 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,
1307 **L(A) = ∅**. The dimensions of the matrix do not change.

1308 **4.2.3.4 Matrix_nrows: Number of rows in a matrix**

1309 Retrieve the number of rows in a matrix.

1310 **C Syntax**

```
1311         GrB_Info GrB_Matrix_nrows(GrB_Index          *nrows,  
1312                                 const GrB_Matrix  A);
```

1313 **Parameters**

1314 nrows (OUT) On successful return, contains the number of rows in the matrix.

1315 A (IN) An existing GraphBLAS matrix being queried.

1316 **Return Values**

1317 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1318 cessfully and the value of nrows has been set.

1319 GrB_PANIC Unknown internal error.

1320 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1321 GraphBLAS objects (input or output) is in an invalid state caused
1322 by a previous execution error. Call GrB_error() to access any error
1323 messages generated by the implementation.

1324 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
1325 Matrix_new or Matrix_dup.

1326 GrB_NULL_POINTER nrows pointer is NULL.

1327 **Description**

1328 Return **nrows(A)** in **nrows** (the number of rows).

1329 **4.2.3.5 Matrix_ncols: Number of columns in a matrix**

1330 Retrieve the number of columns in a matrix.

1331 **C Syntax**

```
1332         GrB_Info GrB_Matrix_ncols(GrB_Index          *ncols,  
1333                                 const GrB_Matrix  A);
```

1334 **Parameters**

1335 ncols (OUT) On successful return, contains the number of columns in the matrix.

1336 A (IN) An existing GraphBLAS matrix being queried.

1337 **Return Values**

1338 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1339 cessfully and the value of ncols has been set.

1340 GrB_PANIC Unknown internal error.

1341 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1342 GraphBLAS objects (input or output) is in an invalid state caused
1343 by a previous execution error. Call GrB_error() to access any error
1344 messages generated by the implementation.

1345 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
1346 Matrix_new or Matrix_dup.

1347 GrB_NULL_POINTER ncols pointer is NULL.

1348 **Description**

1349 Return `ncols(A)` in `ncols` (the number of columns).

1350 **4.2.3.6 Matrix_nvals: Number of stored elements in a matrix**

1351 Retrieve the number of stored elements (tuples) in a matrix.

1352 **C Syntax**

```
1353           GrB_Info GrB_Matrix_nvals(GrB_Index            *nvals,  
1354                                    const GrB_Matrix A);
```

1355 **Parameters**

1356 nvals (OUT) On successful return, contains the number of stored elements (tuples) in
1357 the matrix.

1358 A (IN) An existing GraphBLAS matrix being queried.

1359 **Return Values**

1360 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1361 cessfully and the value of `nvals` has been set.

1362 GrB_PANIC Unknown internal error.

1363 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1364 GraphBLAS objects (input or output) is in an invalid state caused
1365 by a previous execution error. Call `GrB_error()` to access any error
1366 messages generated by the implementation.

1367 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1368 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
1369 `Matrix_new` or `Matrix_dup`.

1370 GrB_NULL_POINTER The `nvals` pointer is `NULL`.

1371 **Description**

1372 Return `nvals(A)` in `nvals`. This is the number of tuples stored in matrix `A`, which is the size of
1373 `L(A)` (see Section 3.5).

1374 **4.2.3.7 Matrix_build: Store elements from tuples into a matrix**

1375 **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);
```

1376 **Parameters**

1377 C (INOUT) An existing Matrix object to store the result.

1378 row_indices (IN) Pointer to an array of row indices.

1379 col_indices (IN) Pointer to an array of column indices.

1380 values (IN) Pointer to an array of scalars of a type that is compatible with the domain of
1381 matrix, `C`.

1382 n (IN) The number of entries contained in each array (the same for `row_indices`,
1383 `col_indices`, and `values`).

1384 dup (IN) An associative and commutative binary function to apply when duplicate
1385 values for the same location are present in the input arrays. All three domains of
1386 dup must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$.

1387 Return Values

1388 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1389 blocking mode, this indicates that the API checks for the input
1390 arguments passed successfully. Either way, output matrix C is ready
1391 to be used in the next method of the sequence.

1392 GrB_PANIC Unknown internal error.

1393 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1394 GraphBLAS objects (input or output) is in an invalid state caused
1395 by a previous execution error. Call `GrB_error()` to access any error
1396 messages generated by the implementation.

1397 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1398 GrB_UNINITIALIZED_OBJECT Either C has not been initialized by a call to `GrB_Matrix_new` or
1399 by `GrB_Matrix_dup`, or dup has not been initialized by a call to `GrB_BinaryOp_new`.
1400

1401 GrB_NULL_POINTER `row_indices`, `col_indices` or `values` pointer is NULL.

1402 GrB_INDEX_OUT_OF_BOUNDS A value in `row_indices` or `col_indices` is outside the allowed range for
1403 C.

1404 GrB_DOMAIN_MISMATCH Either the domains of the GraphBLAS binary operator `dup` are not
1405 all the same, or the domains of `values` and C are incompatible with
1406 each other or D_{dup} .

1407 GrB_OUTPUT_NOT_EMPTY Output matrix C already contains valid tuples (elements). In other
1408 words, `GrB_Matrix_nvals(C)` returns a positive value.

1409 Description

1410 An internal matrix $\tilde{C} = \langle D_{dup}, \mathbf{nrows}(C), \mathbf{ncols}(C), \emptyset \rangle$ is created, which only differs from C in its
1411 domain.

1412 Each tuple $\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the output
1413 in the form of

$$\tilde{C}(\text{row_indices}[k], \text{col_indices}[k]) = (D_{dup}) \text{values}[k].$$

1414 If multiple values for the same location are present in the input arrays, the `dup` binary operand is
 1415 used to reduce them before assignment into $\tilde{\mathbf{C}}$ as follows:

$$1416 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row_indices}[k]=i \wedge \text{col_indices}[k]=j} (D_{dup}) \text{values}[k],$$

1417 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{C}}$ is copied into \mathbf{C} via typecasting its
 1418 values to $\mathbf{D}(\mathbf{C})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

1419 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

1420 It is an error to call this function on an output object with existing elements. In other words,
 1421 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

1422 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,
 1423 `col_indices`, or `values`.

1424 **4.2.3.8 Matrix_setElement: Set a single element in matrix**

1425 Set one element of a matrix to a given value.

1426 **C Syntax**

```
1427     GrB_Info GrB_Matrix_setElement(GrB_Matrix  C,
1428                                 <type>     val,
1429                                 GrB_Index   row_index,
1430                                 GrB_Index   col_index);
```

1431 **Parameters**

1432 `C` (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.

1433 `val` (IN) Scalar value to assign. The type must be compatible with the domain of `C`.

1434 `row_index` (IN) Row index of element to be assigned

1435 `col_index` (IN) Column index of element to be assigned

1436 **Return Values**

1437 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 1438 blocking mode, this indicates that the compatibility tests on in-
 1439 dex/dimensions and domains for the input arguments passed suc-
 1440 cessfully. Either way, the output matrix `C` is ready to be used in
 1441 the next method of the sequence.

1442 GrB_PANIC Unknown internal error.

1443 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1444 GraphBLAS objects (input or output) is in an invalid state caused
1445 by a previous execution error. Call GrB_error() to access any error
1446 messages generated by the implementation.

1447 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1448 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to
1449 Matrix_new or Matrix_dup.

1450 GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e., not less
1451 than nrows(C) or ncols(C), respectively).

1452 GrB_DOMAIN_MISMATCH The domains of C and val are incompatible.

1453 **Description**

1454 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** must be
1455 compatible with **D(C)**. Two domains are compatible with each other if values from one domain can
1456 be cast to values in the other domain as per the rules of the C language. In particular, domains from
1457 Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible
1458 with itself. If any compatibility rule above is violated, execution of GrB_Matrix_extractElement ends
1459 and the domain mismatch error listed above is returned.

1460 Then, both index parameters are checked for valid values where following conditions must hold:

$$1461 \qquad 0 \leq \text{row_index} < \text{nrows}(C),$$

$$\qquad 0 \leq \text{col_index} < \text{ncols}(C)$$

1462 If either of these conditions is violated, execution of GrB_Matrix_extractElement ends and the invalid
1463 index error listed above is returned.

1464 We are now ready to carry out the assignment of val; that is,

$$1465 \qquad C(\text{row_index}, \text{col_index}) = \text{val}$$

1466 If a value existed at this location in C, it will be overwritten; otherwise, and new value is stored in
1467 C.

1468 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new contents
1469 of C is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits with
1470 return value GrB_SUCCESS and the new content of vector C is as defined above but may not be
1471 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1472 **4.2.3.9 Matrix_extractElement: Extract a single element from a matrix**

1473 Extract one element of a matrix into a scalar.

1474 **C Syntax**

```
1475     GrB_Info GrB_Matrix_extractElement(<type>      *val,  
1476                                     const GrB_Matrix A,  
1477                                     GrB_Index      row_index,  
1478                                     GrB_Index      col_index);  
1479
```

1480 **Parameters**

1481 **val** (OUT) Pointer to a scalar of type that is compatible with the domain of matrix **A**.
1482 On successful return, this scalar holds the result of the operation. Any previous
1483 value in **val** is overwritten.

1484 **A** (IN) The GraphBLAS matrix from which an element is extracted.

1485 **row_index** (IN) The row index of location in **A** to extract.

1486 **col_index** (IN) The column index of location in **A** to extract.

1487 **Return Values**

1488 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1489 cessfully. This indicates that the compatibility tests on dimensions
1490 and domains for the input arguments passed successfully, and the
1491 output scalar, **val**, has been computed and is ready to be used in
1492 the next method of the sequence.

1493 **GrB_PANIC** Unknown internal error.

1494 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1495 GraphBLAS objects (input or output) is in an invalid state caused
1496 by a previous execution error. Call **GrB_error()** to access any error
1497 messages generated by the implementation.

1498 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1499 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to
1500 **Matrix_new** or **Matrix_dup**.

1501 **GrB_NULL_POINTER** **val** pointer is NULL.

1502 **GrB_NO_VALUE** There is no stored value at specified location.

1503 **GrB_INVALID_INDEX** **row_index** or **col_index** is outside the allowable range (i.e. less than
1504 zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-
1505 tively).

1506 **GrB_DOMAIN_MISMATCH** The domains of the matrix and scalar are incompatible.

1507 **Description**

1508 First, the scalar and input matrix are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ must be
1509 compatible with $\mathbf{D}(A)$. Two domains are compatible with each other if values from one domain can
1510 be cast to values in the other domain as per the rules of the C language. In particular, domains from
1511 Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible
1512 with itself. If any compatibility rule above is violated, execution of `GrB_Matrix_extractElement` ends
1513 and the domain mismatch error listed above is returned.

1514 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 1515 \quad & 0 \leq \text{row_index} < \mathbf{nrows}(A), \\ & 0 \leq \text{col_index} < \mathbf{ncols}(A) \end{aligned}$$

1516 If either of these conditions is violated, execution of `GrB_Matrix_extractElement` ends and the invalid
1517 index error listed above is returned.

1518 We are now ready to carry out the extract into the output argument, `val`; that is,

$$1519 \quad \text{val} = A(\text{row_index}, \text{col_index})$$

1520 where the following condition must be true:

$$1521 \quad (\text{row_index}, \text{col_index}) \in \mathbf{ind}(A)$$

1522 If this condition is violated, execution of `GrB_Matrix_extractElement` ends and the "no value" error
1523 listed above is returned.

1524 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1525 `GrB_SUCCESS`, the new contents of `val` are as defined above. In other words, the method does not
1526 return until any operations required to fully compute the GraphBLAS matrix `A` have completed.

1527 In `GrB_NONBLOCKING` mode, if the return value is other than `GrB_SUCCESS`, an error in a method
1528 occurring earlier in the sequence may have occurred that prevents completion of the GraphBLAS
1529 matrix `A`. The `GrB_error()` method should be called for additional information about such errors.

1530 **4.2.3.10 Matrix_extractTuples: Extract tuples from a matrix**

1531 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

1532 **C Syntax**

```
1533     GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,  
1534                                     GrB_Index      *col_indices,  
1535                                     <type>         *values,  
1536                                     GrB_Index      *n,  
1537                                     const GrB_Matrix A);
```

1538 **Parameters**

1539 `row_indices` (OUT) Pointer to an array of row indices that is large enough to hold all of the
1540 row indices.

1541 `col_indices` (OUT) Pointer to an array of column indices that is large enough to hold all of the
1542 column indices.

1543 `values` (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
1544 the stored values whose type is compatible with $\mathbf{D}(\mathbf{A})$.

1545 `n` (INOUT) Pointer to a value indicating (in input) the number of elements the `values`,
1546 `row_indices`, and `col_indices` arrays can hold. Upon return, it will contain the number
1547 of values written to the arrays.

1548 `A` (IN) An existing GraphBLAS matrix.

1549 **Return Values**

1550 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
1551 cessfully. This indicates that the compatibility tests on the input
1552 argument passed successfully, and the output arrays, `indices` and
1553 `values`, have been computed.

1554 `GrB_PANIC` Unknown internal error.

1555 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1556 GraphBLAS objects (input or output) is in an invalid state caused
1557 by a previous execution error. Call `GrB_error()` to access any error
1558 messages generated by the implementation.

1559 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1560 `GrB_INSUFFICIENT_SPACE` Not enough space in `row_indices`, `col_indices`, and `values` (as indicated
1561 by the `n` parameter) to hold all of the tuples that will be extracted.

1562 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS matrix, `A`, has not been initialized by a call to
1563 `Matrix_new` or `Matrix_dup`.

1564 `GrB_NULL_POINTER` `row_indices`, `col_indices`, `values` or `n` pointer is `NULL`.

1565 `GrB_DOMAIN_MISMATCH` The domains of the `A` matrix and `values` array are incompatible
1566 with one another.

1567 **Description**

1568 This method will extract all the tuples from the GraphBLAS matrix `A`. The values associated with
1569 those tuples are placed in the `values` array, the column indices are placed in the `col_indices` array,

1570 and the row indices are placed in the `row_indices` array. These output arrays are pre-allocated by
1571 the user before calling this function such that each output array has enough space to hold at least
1572 `GrB_Matrix_nvals(A)` elements.

1573 Upon return of this function, a pair of $\{\text{row_indices}[k], \text{col_indices}[k]\}$ are unique for every valid k ,
1574 but they are not required to be sorted in any particular order. Each tuple (i, j, A_{ij}) in `A` is unzipped
1575 and copied into a distinct k th location in output vectors:

$$\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\} \leftarrow (i, j, A_{ij}),$$

1576 where $0 \leq k < \text{GrB_Matrix_nvals}(v)$. No gaps in output vectors are allowed; that is, if `row_indices[k]`,
1577 `col_indices[k]` and `values[k]` exist upon return, so does `row_indices[j]`, `col_indices[j]` and `values[j]` for all
1578 j such that $0 \leq j < k$.

1579 Note that if the value in `n` on input is less than the number of values contained in the matrix `A`,
1580 then a `GrB_INSUFFICIENT_SPACE` error is returned since it is undefined which subset of values
1581 would be extracted.

1582 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1583 `GrB_SUCCESS`, the new contents of the arrays `row_indices`, `col_indices` and `values` are as defined
1584 above. In other words, the method does not return until any operations required to fully compute
1585 the GraphBLAS vector `A` have completed.

1586 In `GrB_NONBLOCKING` mode, if the return value is not `GrB_SUCCESS`, an error in a method
1587 occurring earlier in the sequence may have occurred that prevents completion of the GraphBLAS
1588 vector `A`. The `GrB_error()` method should be called for additional information about these errors.

1589 4.2.4 Descriptor Methods

1590 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-
1591 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

1592 4.2.4.1 Descriptor_new: Create new descriptor

1593 Creates a new (empty or default) descriptor.

1594 C Syntax

```
1595 GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);
```

1596 Parameters

1597 `desc` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1598 descriptor.

1599 **Return Value**

1600 GrB_SUCCESS The method completed successfully.

1601 GrB_PANIC unknown internal error.

1602 GrB_OUT_OF_MEMORY not enough memory available for operation.

1603 GrB_NULL_POINTER desc pointer is NULL.

1604 **Description**

1605 Creates a new descriptor object and returns a handle to it in `desc`. A newly created descriptor can
1606 be populated by calls to `Descriptor_set`.

1607 It is not an error to call this method more than once on the same variable; however, the handle to
1608 the previously created object will be overwritten.

1609 **4.2.4.2 Descriptor_set: Set content of descriptor**

1610 Sets the content for a field for an existing descriptor.

1611 **C Syntax**

```
1612                   GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
1613                                                GrB_Desc_Field        field,  
1614                                                GrB_Desc_Value        val);
```

1615 **Parameters**

1616 desc (IN) An existing GraphBLAS descriptor to be modified.

1617 field (IN) The field being set.

1618 val (IN) New value for the field being set.

1619 **Return Values**

1620 GrB_SUCCESS operation completed successfully.

1621 GrB_PANIC unknown internal error.

1622 GrB_OUT_OF_MEMORY not enough memory available for operation.

1623 GrB_UNINITIALIZED_OBJECT the `desc` parameter has not been initialized by a call to `new`.

1624 GrB_INVALID_VALUE invalid value set on the field, or invalid field.

1625 **Description**

1626 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor
1627 to set the value associated with that field. Valid values for the `field` parameter include the following:

1628 `GrB_OUTP` refers to the output parameter (result) of the operation.

1629 `GrB_MASK` refers to the mask parameter of the operation.

1630 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

1631 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

1632 Valid values for the `val` parameter are:

1633 `GrB_SCMP` Use the structural complement of the corresponding mask (`GrB_MASK`) pa-
1634 parameter.

1635 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input
1636 matrix parameters only).

1637 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear the
1638 matrix first (or clear the non-masked entries). The default behavior is to
1639 leave non-masked locations unchanged. Valid for the `GrB_OUTP` parameter
1640 only.

1641 A value for a given field may be set multiple times. For a sequence of calls to the `GrB_Descriptor_set`
1642 method, the final call encountered in program order overwrites prior values to define the observed
1643 value for that field. Fields that are not set have their default value, as defined in Section 3.7.

1644 **4.2.5 free method**

1645 Destroys a previously created GraphBLAS object and releases any resources associated with the
1646 object.

1647 **C Syntax**

```
1648           GrB_Info GrB_free(GrB_Object *obj);
```

1649 **Parameters**

1650 `obj` (INOUT) An existing GraphBLAS object to be destroyed. Can be any of the
1651 opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,
1652 binary op, unary op, or type. On successful completion of `GrB_free`, `obj` behaves
1653 as an uninitialized object.

1654 **Return Values**

1655 `GrB_SUCCESS` operation completed successfully

1656 `GrB_PANIC` unknown internal error. If this return value is encountered when
1657 in nonblocking mode, the error responsible for the panic condition
1658 could be from any method involved in the computation of the input
1659 object. The `GrB_error()` method should be called for additional
1660 information.

1661 **Description**

1662 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime
1663 system. A call to `GrB_free` frees those resources so they are available for use by other GraphBLAS
1664 objects.

1665 The parameter passed into `GrB_free` is a handle referencing a GraphBLAS opaque object of a data
1666 type from table 2.1. After the `GrB_free` method returns, the object referenced by the input handle
1667 is destroyed and the handle has the value `GrB_INVALID_HANDLE`. The handle can be used in
1668 subsequent GraphBLAS methods but only after the handle has been reinitialized with a call the
1669 the appropriate `_new` or `_dup` method.

1670 Note that unlike other GraphBLAS methods, calling `GrB_free` with an object with an invalid handle
1671 is legal. The system may attempt to free resources that might be associated with that object, if
1672 possible, and return normally.

1673 When using `GrB_free` it is possible to create a dangling reference to an object. This would occur
1674 when a handle is assigned to a second variable of the same opaque type. This creates two handles
1675 that reference the same object. If `GrB_free` is called with one of the variables, the object is destroyed
1676 and the handle associated with the other variable no longer references a valid object. This is not an
1677 error condition that the implementation of the GraphBLAS API can be expected to catch, hence
1678 programmers must take care to prevent this situation from occurring.

1679 **4.3 GraphBLAS Operations**

1680 The GraphBLAS operations are defined in the GraphBLAS math specification and summarized in
1681 Table 4.1. In addition to methods that implement these fundamental GraphBLAS operations, we
1682 support a number of variants that have been found to be especially useful in algorithm development.
1683 A flowchart of the overall behavior of a GraphBLAS operation is shown in Figure 4.1.

1684 **Domains and Casting**

1685 A GraphBLAS operation is only valid when the domains of the GraphBLAS objects are mathemat-
1686 ically consistent. The C programming language defines implicit casts between built-in data types.
1687 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 specified, 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	$\mathbf{C}\langle\mathbf{M}, z\rangle$	$= \mathbf{C} \odot \mathbf{A}(i, j)$
	$\mathbf{w}\langle\mathbf{m}, z\rangle$	$= \mathbf{w} \odot \mathbf{u}(i)$
assign	$\mathbf{C}\langle\mathbf{M}, z\rangle(i, j)$	$= \mathbf{C}(i, j) \odot \mathbf{A}$
	$\mathbf{w}\langle\mathbf{m}, z\rangle(i)$	$= \mathbf{w}(i) \odot \mathbf{u}$

1688 casts. It is the responsibility of the user to assure that these casts are appropriate for the algorithm
1689 in question. For example, a cast to int implies truncation of a floating point type. Depending on
1690 the operation, this truncation error could lead to erroneous results. Furthermore, casting a wider
1691 type onto a narrower type can lead to overflow errors. The GraphBLAS operations do not attempt
1692 to protect a user from these sorts of errors.

1693 When user-define types are involved, however, GraphBLAS requires strict equivalence between
1694 types and no casting is supported. If GraphBLAS detects these mismatches, it will return a
1695 domain mismatch error.

1696 Dimensions and Transposes

1697 GraphBLAS operations also make assumptions about the numbers of dimensions and sizes of vectors
1698 and matrices in an operation. An operation will test these sizes and report an error if they are not
1699 *shape compatible*. For example, when multiplying two matrices, $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, the number of rows

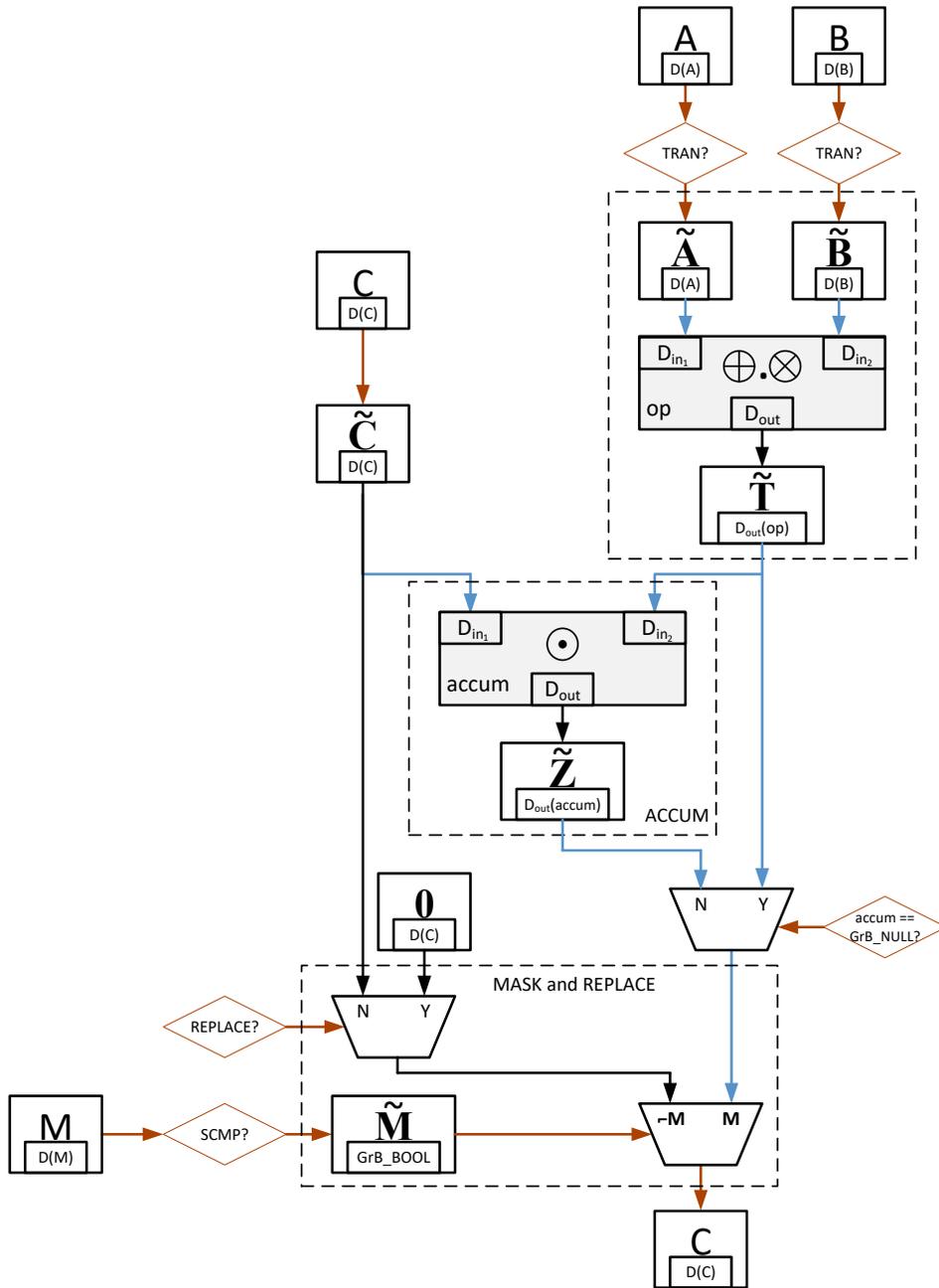


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. Orange arrows denote where “as if copy” takes place (including both collections and descriptor settings).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

1700 of \mathbf{C} must equal the number of rows of \mathbf{A} , the number of columns of \mathbf{A} must match the number of
1701 rows of \mathbf{B} , and the number of columns of \mathbf{C} must match then number of columns of \mathbf{B} . This is the
1702 behavior expected given the mathematical definition of the operations.

1703 For most of the GraphBLAS operations involving matrices, an optional descriptor can modify the
1704 matrix associated with an input GraphBLAS matrix object. For example, if an input matrix is an
1705 argument to a GraphBLAS operation and the associated descriptor indicates the transpose option,
1706 then the operation occurs as if on the transposed matrix. In this case, the relationships between
1707 the sizes in each dimension shift in the mathematically expected way.

1708 **Masks and Structural Complements**

1709 When a GraphBLAS operation supports the use of an optional mask, that mask is specified through
1710 a GraphBLAS vector (for one-dimensional masks) or a GraphBLAS matrix (for two-dimensional
1711 masks). When a mask is used, it is applied to the result from the operation wherever the mask
1712 evaluates to true, and then that result is either assigned to the provided output matrix/vector or,
1713 if a binary accumulation operation is provided, the result is accumulated into the corresponding
1714 elements of the provided output matrix/vector.

1715 Given a GraphBLAS vector $\mathbf{v} = \langle D, N, \{(i, v_i)\} \rangle$, a one-dimensional mask $\mathbf{m} = \langle N, \{i : (\text{bool})v_i =$
1716 $\text{true}\} \rangle$ is derived for use in the operation, where $(\text{bool})v_i$ denotes casting the value v_i to a Boolean
1717 value (true or false).

1718 Given a GraphBLAS matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$, a two-dimensional mask $\mathbf{M} = \langle M, N, \{(i, j) :$
1719 $(\text{bool})A_{ij} = \text{true}\} \rangle$ is derived for use in the operation, where $(\text{bool})A_{ij}$ denotes casting the value
1720 A_{ij} to a Boolean value (true or false).

1721 In both the one- and two-dimensional cases, the mask may go through a structural complement
1722 operation (§ 3.6) as specified in the descriptor, before a final mask is generated for use in the
1723 operation.

1724 When the descriptor of an operation with a mask has specified that the `GrB.REPLACE` value is
1725 to be applied to the output (`GrB.OUTPUT`), then anywhere the mask is not true, the corresponding
1726 location in the output is cleared.

1727 **Invalid and uninitialized objects**

1728 Upon entering a GraphBLAS operation, the first step is a check that all objects are valid and
1729 initialized. (Optional parameters can be set to `GrB.NULL`, which always counts as a valid object.)
1730 An invalid object is one that could not be computed due to some previous execution error. An
1731 uninitialized object is one that has not yet been created by a corresponding `new` or `dup` method.
1732 Appropriate error codes are returned if an object is not initialized (`GrB.UNINITIALIZED_OBJECT`)
1733 or invalid (`GrB.INVALID_OBJECT`).

1734 To support the detection of as many cases of uninitialized objects as possible, it is strongly recom-
1735 mended to initialize all GraphBLAS objects to the predefined value `GrB.INVALID_HANDLE` at the
1736 point of their declaration, as shown in the following examples:

```

1737         GrB_Type          type = GrB_INVALID_HANDLE;
1738         GrB_Semiring      semiring = GrB_INVALID_HANDLE;
1739         GrB_Matrix        matrix = GrB_INVALID_HANDLE;

```

1740 Compliance

1741 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.
 1742 That is, for each operation we give a recipe for producing its outcome. It should be understood
 1743 that any implementation that produces the same outcome, and follows the GraphBLAS execution
 1744 model (§ 2.8) and error model (§ 2.9), is a conforming implementation.

1745 4.3.1 mxm: Matrix-matrix multiply

1746 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

1747 C Syntax

```

1748         GrB_Info GrB_mxm(GrB_Matrix          C,
1749                         const GrB_Matrix    Mask,
1750                         const GrB_BinaryOp   accum,
1751                         const GrB_Semiring   op,
1752                         const GrB_Matrix    A,
1753                         const GrB_Matrix    B,
1754                         const GrB_Descriptor desc);

```

1755 Parameters

1756 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 1757 that may be accumulated with the result of the matrix product. On output, the
 1758 matrix holds the results of the operation.

1759 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 1760 stored into the output matrix C. The mask dimensions must match those of the
 1761 matrix C and the domain of the Mask matrix must be of type bool or any of the
 1762 predefined “built-in” types in Table 2.2. If the default matrix is desired (i.e., with
 1763 correct dimensions and filled with true), GrB_NULL should be specified.

1764 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 1765 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 1766 specified.

1767 **op** (IN) The semiring used in the matrix-matrix multiply.

1768 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 1769 multiplication.

1770 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 1771 multiplication.

1772 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 1773 should be specified. Non-default field/value pairs are listed as follows:
 1774

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.

1775

1776 Return Values

1777 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 1778 blocking mode, this indicates that the compatibility tests on di-
 1779 mensions and domains for the input arguments passed successfully.
 1780 Either way, output matrix C is ready to be used in the next method
 1781 of the sequence.

1782 GrB_PANIC Unknown internal error.

1783 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 1784 GraphBLAS objects (input or output) is in an invalid state caused
 1785 by a previous execution error. Call GrB_error() to access any error
 1786 messages generated by the implementation.

1787 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

1788 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 1789 a call to new (or Matrix_dup for matrix parameters).

1790 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

1791 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 1792 corresponding domains of the semiring or accumulation operator,
 1793 or the mask's domain is not compatible with bool.

1794 Description

1795 GrB_mxm computes the matrix product $C = A \otimes . \oplus B$ or, if an optional binary accumulation operator
 1796 (\odot) is provided, $C = C \odot (A \otimes . \oplus B)$ (where matrices A and B can be optionally transposed).
 1797 Logically, this operation occurs in three steps:

1798 **Setup** The internal matrices and mask used in the computation are formed and their domains
 1799 and dimensions are tested for compatibility.

1800 **Compute** The indicated computations are carried out.

1801 **Output** The result is written into the output matrix, possibly under control of a mask.

1802 Up to four argument matrices are used in the `GrB_mxm` operation:

- 1803 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 1804 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 1805 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 1806 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

1807 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for
 1808 domain compatibility as follows:

- 1809 1. The domain of `Mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 1810 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 1811 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 1812 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 1813 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 1814 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 1815 of the accumulation operator.

1816 Two domains are compatible with each other if values from one domain can be cast to values in
 1817 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
 1818 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 1819 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch
 1820 error listed above is returned.

1821 From the argument matrices, the internal matrices and mask used in the computation are formed
 1822 (\leftarrow denotes copy):

- 1823 1. Matrix $\tilde{C} \leftarrow C$.
- 1824 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:
 - 1825 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
 1826 $j < \mathbf{ncols}(C)\} \rangle$.
 - 1827 (b) Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) =$
 1828 $\text{true}\} \rangle$.

1829 (c) If desc[GrB_MASK].GrB_SCMP is set, then $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}$.

1830 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

1831 4. Matrix $\widetilde{\mathbf{B}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

1832 The internal matrices and masks are checked for dimension compatibility. The following conditions
1833 must hold:

1834 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.

1835 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.

1836 3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

1837 4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{B}})$.

1838 5. $\mathbf{ncols}(\widetilde{\mathbf{A}}) = \mathbf{nrows}(\widetilde{\mathbf{B}})$.

1839 If any compatibility rule above is violated, execution of GrB_mxm ends and the dimension mismatch
1840 error listed above is returned.

1841 From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with
1842 GrB_SUCCESS return code and defer any computation and/or execution error codes.

1843 We are now ready to carry out the matrix multiplication and any additional associated operations.
1844 We describe this in terms of two intermediate matrices:

1845 • $\widetilde{\mathbf{T}}$: The matrix holding the product of matrices $\widetilde{\mathbf{A}}$ and $\widetilde{\mathbf{B}}$.

1846 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

1847 The intermediate matrix $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{B}}), \{(i, j, T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:, j)) \neq \emptyset\} \rangle$ is created. The value of each of its elements is computed by

1849
$$T_{ij} = \bigoplus_{k \in \mathbf{ind}(\widetilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:, j))} (\widetilde{\mathbf{A}}(i, k) \otimes \widetilde{\mathbf{B}}(k, j)),$$

1850 where \oplus and \otimes are the additive and multiplicative operators of semiring op , respectively.

1851 The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

1852 • If $\text{accum} = \text{GrB_NULL}$, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.

1853 • If accum is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

1854
$$\widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

1855 The values of the elements of $\widetilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
1856 indices in $\widetilde{\mathbf{C}}$ and $\widetilde{\mathbf{T}}$.

1857
$$Z_{ij} = \widetilde{\mathbf{C}}(i, j) \odot \widetilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}})),$$

1858

1859

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

1860

1861

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

1862

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

1863

1864

1865

Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{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”.

1866

1867

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

1868

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

1869

1870

1871

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

1872

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

1873

1874

1875

1876

1877

In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{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 \mathbf{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.

1878

4.3.2 vxm: Vector-matrix multiply

1879

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

1880

C Syntax

1881

1882

1883

1884

1885

1886

1887

```
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);
```

1888 **Parameters**

1889 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 1890 that may be accumulated with the result of the vector-matrix product. On output,
 1891 this vector holds the results of the operation.

1892 **mask** (IN) An optional “write” mask that controls which results from this operation are
 1893 stored into the output vector **w**. The mask dimensions must match those of the
 1894 vector **w** and the domain of the **mask** vector must be of type **bool** or any of the
 1895 predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with
 1896 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

1897 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
 1898 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 1899 specified.

1900 **op** (IN) Semiring used in the vector-matrix multiply.

1901 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 1902 multiplication.

1903 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 1904 multiplication.

1905 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 1906 should be specified. Non-default field/value pairs are listed as follows:

1907

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.

1908

1909 **Return Values**

1910 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 1911 blocking mode, this indicates that the compatibility tests on di-
 1912 mensions and domains for the input arguments passed successfully.
 1913 Either way, output vector **w** is ready to be used in the next method
 1914 of the sequence.

1915 **GrB_PANIC** Unknown internal error.

1916 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 1917 GraphBLAS objects (input or output) is in an invalid state caused

1918 by a previous execution error. Call `GrB_error()` to access any error
1919 messages generated by the implementation.

1920 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

1921 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by
1922 a call to `new` (or `dup` for matrix or vector parameters).

1923 `GrB_DIMENSION_MISMATCH` Mask, vector, and/or matrix dimensions are incompatible.

1924 `GrB_DOMAIN_MISMATCH` The domains of the various vectors/matrices are incompatible with
1925 the corresponding domains of the semiring or accumulation operator,
1926 or the mask's domain is not compatible with `bool`.

1927 Description

1928 `GrB_vxm` computes the vector-matrix product $\mathbf{w}^T = \mathbf{u}^T \otimes \oplus \mathbf{A}$, or, if an optional binary accumulation
1929 operator (\odot) is provided, $\mathbf{w}^T = \mathbf{w}^T \odot (\mathbf{u}^T \otimes \oplus \mathbf{A})$ (where matrix \mathbf{A} can be optionally transposed).
1930 Logically, this operation occurs in three steps:

1931 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
1932 domains/dimensions are tested for compatibility.

1933 **Compute** The indicated computations are carried out.

1934 **Output** The result is written into the output vector, possibly under control of a mask.

1935 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 1936 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 1937 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 1938 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 1939 4. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

1940 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
1941 tested for domain compatibility as follows:

- 1942 1. The domain of `mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 1943 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 1944 3. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 1945 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.

1946 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 1947 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 1948 of the accumulation operator.

1949 Two domains are compatible with each other if values from one domain can be cast to values in
 1950 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
 1951 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 1952 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch
 1953 error listed above is returned.

1954 From the argument vectors and matrices, the internal matrices and mask used in the computation
 1955 are formed (\leftarrow denotes copy):

- 1956 1. Vector $\tilde{\mathbf{w}} \leftarrow w$.
- 1957 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 1958 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.
 - 1959 (b) Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 1960 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 1961 3. Vector $\tilde{\mathbf{u}} \leftarrow u$.
- 1962 4. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? A^T : A$.

1963 The internal matrices and masks are checked for shape compatibility. The following conditions
 1964 must hold:

- 1965 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.
- 1966 2. $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$.
- 1967 3. $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$.

1968 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch
 1969 error listed above is returned.

1970 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 1971 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

1972 We are now ready to carry out the vector-matrix multiplication and any additional associated
 1973 operations. We describe this in terms of two intermediate vectors:

- 1974 • $\tilde{\mathbf{t}}$: The vector holding the product of vector $\tilde{\mathbf{u}}^T$ and matrix $\tilde{\mathbf{A}}$.
- 1975 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

1976 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(j, t_j) : \mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{A}}(:, j)) \neq \emptyset\} \rangle$ is created.
 1977 The value of each of its elements is computed by

$$1978 \quad t_j = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

1979 where \oplus and \otimes are the additive and multiplicative operators of semiring \mathbf{op} , respectively.

1980 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 1981 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 1982 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$1983 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

1984 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 1985 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 1986 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 1987 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 1988 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 1989 \end{aligned}$$

1990 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

1992 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 1993 using what is called a *standard vector mask and replace*. This is carried out under control of the
 1994 mask which acts as a “write mask”.

- 1995 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 1996 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$1997 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 1998 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 1999 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 2000 mask are unchanged:

$$2001 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

2002 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 2003 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 2004 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 2005 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

2006 4.3.3 mxv: Matrix-vector multiply

2007 Multiplies a matrix by a vector on a semiring. The result is a vector.

2008 **C Syntax**

```

2009         GrB_Info GrB_mxv(GrB_Vector          w,
2010                         const GrB_Vector     mask,
2011                         const GrB_BinaryOp    accum,
2012                         const GrB_Semiring    op,
2013                         const GrB_Matrix     A,
2014                         const GrB_Vector     u,
2015                         const GrB_Descriptor  desc);

```

2016 **Parameters**

2017 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 2018 that may be accumulated with the result of the matrix-vector product. On output,
 2019 this vector holds the results of the operation.

2020 **mask** (IN) An optional “write” mask that controls which results from this operation are
 2021 stored into the output vector **w**. The mask dimensions must match those of the
 2022 vector **w** and the domain of the **mask** vector must be of type **bool** or any of the
 2023 predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with
 2024 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

2025 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
 2026 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 2027 specified.

2028 **op** (IN) Semiring used in the vector-matrix multiply.

2029 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 2030 multiplication.

2031 **u** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 2032 multiplication.

2033 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 2034 should be specified. Non-default field/value pairs are listed as follows:
 2035

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.

2036

2037 **Return Values**

2038 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
2039 blocking mode, this indicates that the compatibility tests on di-
2040 mensions and domains for the input arguments passed successfully.
2041 Either way, output vector w is ready to be used in the next method
2042 of the sequence.

2043 **GrB_PANIC** Unknown internal error.

2044 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
2045 GraphBLAS objects (input or output) is in an invalid state caused
2046 by a previous execution error. Call `GrB_error()` to access any error
2047 messages generated by the implementation.

2048 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

2049 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
2050 a call to `new` (or `dup` for matrix or vector parameters).

2051 **GrB_DIMENSION_MISMATCH** Mask, vector, and/or matrix dimensions are incompatible.

2052 **GrB_DOMAIN_MISMATCH** The domains of the various vectors/matrices are incompatible with
2053 the corresponding domains of the semiring or accumulation opera-
2054 tor, or the mask's domain is not compatible with `bool`.

2055 **Description**

2056 **GrB_m xv** computes the matrix-vector product $w = A \otimes . \oplus u$, or, if an optional binary accumulation
2057 operator (\odot) is provided, $w = w \odot (A \otimes . \oplus u)$ (where matrix A can be optionally transposed).
2058 Logically, this operation occurs in three steps:

2059 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
2060 domains/dimensions are tested for compatibility.

2061 **Compute** The indicated computations are carried out.

2062 **Output** The result is written into the output vector, possibly under control of a mask.

2063 Up to four argument vectors or matrices are used in the `GrB_m xv` operation:

- 2064 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
2065 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
2066 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
2067 4. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

2068 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
 2069 tested for domain compatibility as follows:

- 2070 1. The domain of `mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 2071 2. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 2072 3. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 2073 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.
- 2074 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
 2075 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$
 2076 of the accumulation operator.

2077 Two domains are compatible with each other if values from one domain can be cast to values in
 2078 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
 2079 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 2080 If any compatibility rule above is violated, execution of `GrB_mvx` ends and the domain mismatch
 2081 error listed above is returned.

2082 From the argument vectors and matrices, the internal matrices and mask used in the computation
 2083 are formed (\leftarrow denotes copy):

- 2084 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 2085 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 2086 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 2087 (b) Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 2088 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 2089 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 2090 4. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

2091 The internal matrices and masks are checked for shape compatibility. The following conditions
 2092 must hold:

- 2093 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$.
- 2094 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 2095 3. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

2096 If any compatibility rule above is violated, execution of `GrB_mvx` ends and the dimension mismatch
 2097 error listed above is returned.

2098 From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with
 2099 GrB_SUCCESS return code and defer any computation and/or execution error codes.

2100 We are now ready to carry out the matrix-vector multiplication and any additional associated
 2101 operations. We describe this in terms of two intermediate vectors:

- 2102 • $\tilde{\mathbf{t}}$: The vector holding the product of matrix $\tilde{\mathbf{A}}$ and vector $\tilde{\mathbf{u}}$.
- 2103 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

2104 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \{(i, t_i) : \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{u}}) \neq \emptyset\} \rangle$ is created.
 2105 The value of each of its elements is computed by

$$2106 \quad t_i = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

2107 where \oplus and \otimes are the additive and multiplicative operators of semiring \mathbf{op} , respectively.

2108 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 2109 • If $\mathbf{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 2110 • If \mathbf{accum} is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$2111 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

2112 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 2113 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 2114 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 2115 \\ 2116 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 2117 \\ 2118 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

2119 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

2120 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 2121 using what is called a *standard vector mask and replace*. This is carried out under control of the
 2122 mask which acts as a “write mask”.

- 2123 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in \mathbf{w} on input to this operation are
 2124 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$2125 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 2126 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 2127 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 2128 mask are unchanged:

$$2129 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

2130 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of
 2131 vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits
 2132 with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but may not
 2133 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

2134 4.3.4 eWiseMult: Element-wise multiplication

2135 **Note:** The difference between eWiseAdd and eWiseMult is not about the element-wise operation
2136 but how the index sets are treated. eWiseAdd returns an object whose indices are the “union” of
2137 the indices of the inputs whereas eWiseMult returns an object whose indices are the “intersection”
2138 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
2139 the set of values from the resulting index set.

2140 4.3.4.1 eWiseMult: Vector variant

2141 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-
2142 ducing a third vector as result.

2143 C Syntax

```
2144     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
2145                          const GrB_Vector mask,  
2146                          const GrB_BinaryOp accum,  
2147                          const GrB_Semiring op,  
2148                          const GrB_Vector u,  
2149                          const GrB_Vector v,  
2150                          const GrB_Descriptor desc);
```

```
2151     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
2152                          const GrB_Vector mask,  
2153                          const GrB_BinaryOp accum,  
2154                          const GrB_Monoid op,  
2155                          const GrB_Vector u,  
2156                          const GrB_Vector v,  
2157                          const GrB_Descriptor desc);
```

```
2159     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
2160                          const GrB_Vector mask,  
2161                          const GrB_BinaryOp accum,  
2162                          const GrB_BinaryOp op,  
2163                          const GrB_Vector u,  
2164                          const GrB_Vector v,  
2165                          const GrB_Descriptor desc);
```

2167 Parameters

2168 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
2169 that may be accumulated with the result of the element-wise operation. On output,
2170 this vector holds the results of the operation.

2171 mask (IN) An optional “write” mask that controls which results from this operation are
 2172 stored into the output vector w . The mask dimensions must match those of the
 2173 vector w and the domain of the `mask` vector must be of type `bool` or any of the
 2174 predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with
 2175 correct dimensions and filled with `true`), `GrB_NULL` should be specified.

2176 accum (IN) An optional binary operator used for accumulating entries into existing w
 2177 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 2178 specified.

2179 op (IN) The semiring, monoid, or binary operator used in the element-wise “product”
 2180 operation. Depending on which type is passed, the following defines the binary
 2181 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$, used:

2182 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

2183 Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ig-
 2184 nored.

2185 Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$; the additive monoid
 2186 is ignored.

2187 u (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 2188 operation.

2189 v (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 2190 operation.

2191 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 2192 should be specified. Non-default field/value pairs are listed as follows:

2193

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 <code>mask</code> .

2194

2195 Return Values

2196 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 2197 blocking mode, this indicates that the compatibility tests on di-
 2198 mensions and domains for the input arguments passed successfully.
 2199 Either way, output vector w is ready to be used in the next method
 2200 of the sequence.

2201 GrB_PANIC Unknown internal error.

2202 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 2203 GraphBLAS objects (input or output) is in an invalid state caused
 2204 by a previous execution error. Call `GrB_error()` to access any error
 2205 messages generated by the implementation.

2206 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2207 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 2208 a call to `new` (or `dup` for vector parameters).

2209 GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

2210 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 2211 responding domains of the binary operator (`op`) or accumulation
 2212 operator, or the mask’s domain is not compatible with `bool`.

2213 Description

2214 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:
 2215 $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$.
 2216 Logically, this operation occurs in three steps:

2217 **Setup** The internal vectors and mask used in the computation are formed and their domains
 2218 and dimensions are tested for compatibility.

2219 **Compute** The indicated computations are carried out.

2220 **Output** The result is written into the output vector, possibly under control of a mask.

2221 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 2222 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 2223 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 2224 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 2225 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

2226 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)
 2227 are tested for domain compatibility as follows:

- 2228 1. The domain of `mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 2229 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 2230 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 2231 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.

2232 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 2233 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 2234 the accumulation operator.

2235 Two domains are compatible with each other if values from one domain can be cast to values
 2236 in the other domain as per the rules of the C language. In particular, domains from Table 2.2
 2237 are all compatible with each other. A domain from a user-defined type is only compatible with
 2238 itself. If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain
 2239 mismatch error listed above is returned.

2240 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 2241 denotes copy):

- 2242 1. Vector $\tilde{w} \leftarrow w$.
- 2243 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
 - 2244 (a) If `mask = GrB_NULL`, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.
 - 2245 (b) Otherwise, $\tilde{m} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 - 2246 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{m} \leftarrow \neg\tilde{m}$.
- 2247 3. Vector $\tilde{u} \leftarrow u$.
- 2248 4. Vector $\tilde{v} \leftarrow v$.

2249 The internal vectors and mask are checked for dimension compatibility. The following conditions
 2250 must hold:

- 2251 1. $\text{size}(\tilde{w}) = \text{size}(\tilde{m}) = \text{size}(\tilde{u}) = \text{size}(\tilde{v})$.

2252 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 2253 mismatch error listed above is returned.

2254 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 2255 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2256 We are now ready to carry out the element-wise “product” and any additional associated operations.
 2257 We describe this in terms of two intermediate vectors:

- 2258 • \tilde{t} : The vector holding the element-wise “product” of \tilde{u} and vector \tilde{v} .
- 2259 • \tilde{z} : The vector holding the result after application of the (optional) accumulation operator.

2260 The intermediate vector $\tilde{t} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{u}), \mathbf{L}(\tilde{t}) = \{(i, t_i) : \text{ind}(\tilde{u}) \cap \text{ind}(\tilde{v}) \neq \emptyset\} \rangle$ is created.
 2261 The value of each of its elements is computed by:

$$2262 \quad t_i = (\tilde{u}(i) \otimes \tilde{v}(i)), \forall i \in (\text{ind}(\tilde{u}) \cap \text{ind}(\tilde{v}))$$

2263 The intermediate vector \tilde{z} is created as follows, using what is called a *standard vector accumulate*:

2264 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

2265 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$2266 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

2267 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 2268 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$2269 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$2270 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$2271 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

2272 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

2273 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 2274 using what is called a *standard vector mask and replace*. This is carried out under control of the
 2275 mask which acts as a “write mask”.

2276 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 2277 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$2278 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

2280 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 2281 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 2282 mask are unchanged:

$$2283 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

2284 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 2285 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 2286 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 2287 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

2288 4.3.4.2 eWiseMult: Matrix variant

2289 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-
 2290 ducing a third matrix as result.

2291 C Syntax

```
2292     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
2293                          const GrB_Matrix Mask,
2294                          const GrB_BinaryOp accum,
2295                          const GrB_Semiring op,
```

```

2297         const GrB_Matrix      A,
2298         const GrB_Matrix      B,
2299         const GrB_Descriptor   desc);
2300
2301     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
2302         const GrB_Matrix      Mask,
2303         const GrB_BinaryOp    accum,
2304         const GrB_Monoid      op,
2305         const GrB_Matrix      A,
2306         const GrB_Matrix      B,
2307         const GrB_Descriptor   desc);
2308
2309     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
2310         const GrB_Matrix      Mask,
2311         const GrB_BinaryOp    accum,
2312         const GrB_BinaryOp    op,
2313         const GrB_Matrix      A,
2314         const GrB_Matrix      B,
2315         const GrB_Descriptor   desc);

```

2316 Parameters

2317 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
2318 that may be accumulated with the result of the element-wise operation. On output,
2319 the matrix holds the results of the operation.

2320 **Mask** (IN) An optional “write” mask that controls which results from this operation are
2321 stored into the output matrix **C**. The mask dimensions must match those of the
2322 matrix **C** and the domain of the **Mask** matrix must be of type `bool` or any of the
2323 predefined “built-in” types in Table 2.2. If the default matrix is desired (i.e., with
2324 correct dimensions end filled with `true`), `GrB_NULL` should be specified.

2325 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
2326 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
2327 specified.

2328 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
2329 operation. Depending on which type is passed, the following defines the binary
2330 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

2331 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

2332 Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ig-
2333 nored.

2334 Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$; the additive monoid
2335 is ignored.

2336 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
2337 operation.

2338 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
2339 operation.

2340 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB.NULL
2341 should be specified. Non-default field/value pairs are listed as follows:
2342

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.

2344 Return Values

2345 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2346 blocking mode, this indicates that the compatibility tests on di-
2347 mensions and domains for the input arguments passed successfully.
2348 Either way, output matrix C is ready to be used in the next method
2349 of the sequence.

2350 GrB_PANIC Unknown internal error.

2351 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2352 GraphBLAS objects (input or output) is in an invalid state caused
2353 by a previous execution error. Call GrB_error() to access any error
2354 messages generated by the implementation.

2355 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2356 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
2357 a call to new (or Matrix_dup for matrix parameters).

2358 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

2359 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
2360 corresponding domains of the binary operator (**op**) or accumulation
2361 operator, or the mask's domain is not compatible with **bool**.

2362 **Description**

2363 This variant of GrB_eWiseMult computes the element-wise “product” of two GraphBLAS matrices:
2364 $C = A \otimes B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \otimes B)$.
2365 Logically, this operation occurs in three steps:

2366 **Setup** The internal matrices and mask used in the computation are formed and their domains
2367 and dimensions are tested for compatibility.

2368 **Compute** The indicated computations are carried out.

2369 **Output** The result is written into the output matrix, possibly under control of a mask.

2370 Up to four argument matrices are used in the GrB_eWiseMult operation:

- 2371 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 2372 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 2373 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 2374 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

2375 The argument matrices, the “product” operator (**op**), and the accumulation operator (if provided)
2376 are tested for domain compatibility as follows:

- 2377 1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
- 2378 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 2379 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 2380 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 2381 5. If **accum** is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
2382 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of **op** must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
2383 the accumulation operator.

2384 Two domains are compatible with each other if values from one domain can be cast to values
2385 in the other domain as per the rules of the C language. In particular, domains from Table 2.2
2386 are all compatible with each other. A domain from a user-defined type is only compatible with
2387 itself. If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the domain
2388 mismatch error listed above is returned.

2389 From the argument matrices, the internal matrices and mask used in the computation are formed
2390 (\leftarrow denotes copy):

- 2391 1. Matrix $\tilde{C} \leftarrow C$.

2392

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

2393

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

2394

2395

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

2396

2397

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

2398

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

2399

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

2400

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

2401

2402

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

2403

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

2404

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

2405

2406

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.

2407

2408

We are now ready to carry out the element-wise “product” and any additional associated operations.

2409

We describe this in terms of two intermediate matrices:

2410

- $\widetilde{\mathbf{T}}$: The matrix holding the element-wise product of $\widetilde{\mathbf{A}}$ and $\widetilde{\mathbf{B}}$.

2411

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

2412

The intermediate matrix $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}}) \neq \emptyset\} \rangle$

2413

is created. The value of each of its elements is computed by

2414

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

2415

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

2416

- If `accum = GrB.NULL`, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.

2417

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

2418

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

2419

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 $\widetilde{\mathbf{T}}$.

2420

2421

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

2422

2423

2424

2425

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

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

2426

where $\odot = \bigcirc$ (accum), and the difference operator refers to set difference.

2427

2428

2429

Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{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”.

2430

2431

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

2432

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

2433

2434

2435

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

2436

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

2437

2438

2439

2440

2441

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of matrix \mathbf{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 \mathbf{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.

2442

4.3.5 eWiseAdd: Element-wise addition

2443

2444

2445

2446

2447

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.

2448

4.3.5.1 eWiseAdd: Vector variant

2449

2450

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

2451

C Syntax

2452

2453

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

```

2454         const GrB_BinaryOp    accum,
2455         const GrB_Semiring    op,
2456         const GrB_Vector      u,
2457         const GrB_Vector      v,
2458         const GrB_Descriptor  desc);
2459
2460     GrB_Info GrB_eWiseAdd(GrB_Vector      w,
2461                         const GrB_Vector  mask,
2462                         const GrB_BinaryOp accum,
2463                         const GrB_Monoid  op,
2464                         const GrB_Vector  u,
2465                         const GrB_Vector  v,
2466                         const GrB_Descriptor desc);
2467
2468     GrB_Info GrB_eWiseAdd(GrB_Vector      w,
2469                         const GrB_Vector  mask,
2470                         const GrB_BinaryOp accum,
2471                         const GrB_BinaryOp op,
2472                         const GrB_Vector  u,
2473                         const GrB_Vector  v,
2474                         const GrB_Descriptor desc);

```

2475 Parameters

2476 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
2477 that may be accumulated with the result of the element-wise operation. On output,
2478 this vector holds the results of the operation.

2479 **mask** (IN) An optional “write” mask that controls which results from this operation are
2480 stored into the output vector **w**. The mask dimensions must match those of the
2481 vector **w** and the domain of the **mask** vector must be of type **bool** or any of the
2482 predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with
2483 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

2484 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
2485 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
2486 specified.

2487 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
2488 operation. Depending on which type is passed, the following defines the binary
2489 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$, used:

2490 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

2491 Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ig-
2492 nored.

2493 Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus(\text{op}) \rangle$; the multiplicative
 2494 binary op and additive identity are ignored.

2495 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
 2496 operation.

2497 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 2498 operation.

2499 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 2500 should be specified. Non-default field/value pairs are listed as follows:
 2501

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.

2503 **Return Values**

2504 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 2505 blocking mode, this indicates that the compatibility tests on di-
 2506 mensions and domains for the input arguments passed successfully.
 2507 Either way, output vector w is ready to be used in the next method
 2508 of the sequence.

2509 **GrB_PANIC** Unknown internal error.

2510 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 2511 GraphBLAS objects (input or output) is in an invalid state caused
 2512 by a previous execution error. Call **GrB_error()** to access any error
 2513 messages generated by the implementation.

2514 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

2515 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 2516 a call to **new** (or **dup** for vector parameters).

2517 **GrB_DIMENSION_MISMATCH** Mask or vector dimensions are incompatible.

2518 **GrB_DOMAIN_MISMATCH** The domains of the various vectors are incompatible with the cor-
 2519 responding domains of the binary operator (**op**) or accumulation
 2520 operator, or the mask's domain is not compatible with **bool**.

2521 **Description**

2522 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors: $\mathbf{w} =$
2523 $\mathbf{u} \oplus \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$. Logically,
2524 this operation occurs in three steps:

2525 **Setup** The internal vectors and mask used in the computation are formed and their domains
2526 and dimensions are tested for compatibility.

2527 **Compute** The indicated computations are carried out.

2528 **Output** The result is written into the output vector, possibly under control of a mask.

2529 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 2530 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 2531 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 2532 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 2533 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

2534 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are
2535 tested for domain compatibility as follows:

- 2536 1. The domain of `mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 2537 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$.
- 2538 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$.
- 2539 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 2540 5. $\mathbf{D}(\mathbf{u})$ and $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$.
- 2541 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
2542 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of
2543 the accumulation operator.

2544 Two domains are compatible with each other if values from one domain can be cast to values
2545 in the other domain as per the rules of the C language. In particular, domains from Table 2.2
2546 are all compatible with each other. A domain from a user-defined type is only compatible with
2547 itself. If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain
2548 mismatch error listed above is returned.

2549 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
2550 denotes copy):

- 2551 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 2552 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
- 2553 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 2554 (b) Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 2555 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 2556 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 2557 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

2558 The internal vectors and mask are checked for dimension compatibility. The following conditions
2559 must hold:

- 2560 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}}) = \mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{v}})$.

2561 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
2562 mismatch error listed above is returned.

2563 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
2564 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2565 We are now ready to carry out the element-wise “sum” and any additional associated operations.
2566 We describe this in terms of two intermediate vectors:

- 2567 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “sum” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 2568 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

2569 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \mathbf{L}(\tilde{\mathbf{t}}) = \{(i, t_i) : \mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$ is created.
2570 The value of each of its elements is computed by:

$$2571 \quad t_i = (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}))$$

$$2572 \quad t_i = \tilde{\mathbf{u}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) - (\mathbf{ind}(\tilde{\mathbf{v}}) \cap \mathbf{ind}(\tilde{\mathbf{u}})))$$

$$2573 \quad t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{v}}) \cap \mathbf{ind}(\tilde{\mathbf{u}})))$$

2576 where the difference operator in the previous expressions refers to set difference.

2577 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 2578 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 2579 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$2580 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

2581 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 2582 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned}
 2583 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\
 2584 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 2585 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 2586 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 2587 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),
 \end{aligned}$$

2588 where $\odot = \bigodot(\text{accum})$, and the difference operator refers to set difference.

2589 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 2590 using what is called a *standard vector mask and replace*. This is carried out under control of the
 2591 mask which acts as a “write mask”.

- 2592 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 2593 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$2594 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 2595 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 2596 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 2597 mask are unchanged:

$$2598 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

2599 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of
 2600 vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits
 2601 with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but may not
 2602 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

2603 4.3.5.2 eWiseAdd: Matrix variant

2604 Perform element-wise (general) addition on the elements of two matrices, producing a third matrix
 2605 as result.

2606 C Syntax

```

2607     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
2608                          const GrB_Matrix Mask,
2609                          const GrB_BinaryOp accum,
2610                          const GrB_Semiring op,
2611                          const GrB_Matrix A,
2612                          const GrB_Matrix B,
2613                          const GrB_Descriptor desc);
2614
2615     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,

```

```

2616         const GrB_Matrix      Mask,
2617         const GrB_BinaryOp    accum,
2618         const GrB_Monoid      op,
2619         const GrB_Matrix      A,
2620         const GrB_Matrix      B,
2621         const GrB_Descriptor  desc);
2622
2623     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
2624                         const GrB_Matrix  Mask,
2625                         const GrB_BinaryOp accum,
2626                         const GrB_BinaryOp op,
2627                         const GrB_Matrix  A,
2628                         const GrB_Matrix  B,
2629                         const GrB_Descriptor desc);

```

2630 Parameters

2631 **C (INOUT)** An existing GraphBLAS matrix. On input, the matrix provides values
2632 that may be accumulated with the result of the element-wise operation. On output,
2633 the matrix holds the results of the operation.

2634 **Mask (IN)** An optional “write” mask that controls which results from this operation are
2635 stored into the output matrix C. The mask dimensions must match those of the
2636 matrix C and the domain of the Mask matrix must be of type `bool` or any of the
2637 predefined “built-in” types in Table 2.2. If the default matrix is desired (i.e., with
2638 correct dimensions and filled with `true`), `GrB_NULL` should be specified.

2639 **accum (IN)** An optional binary operator used for accumulating entries into existing C
2640 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
2641 specified.

2642 **op (IN)** The semiring, monoid, or binary operator used in the element-wise “sum”
2643 operation. Depending on which type is passed, the following defines the binary
2644 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

2645 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

2646 Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$; the identity element is ig-
2647 nored.

2648 Semiring: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus(\text{op}) \rangle$; the multiplicative
2649 binary op and additive identity are ignored.

2650 **A (IN)** The GraphBLAS matrix holding the values for the left-hand matrix in the
2651 operation.

2652 **B (IN)** The GraphBLAS matrix holding the values for the right-hand matrix in the
2653 operation.

2654
2655
2656

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.

2657

2658 Return Values

2659
2660
2661
2662
2663

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.

2664

GrB_PANIC Unknown internal error.

2665
2666
2667
2668

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.

2669

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2670
2671

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).

2672

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

2673
2674
2675

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.

2676 Description

2677
2678
2679

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

2680
2681

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

2682 **Compute** The indicated computations are carried out.

2683 **Output** The result is written into the output matrix, possibly under control of a mask.

2684 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

2685 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

2686 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)

2687 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

2688 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

2689 The argument matrices, the “sum” operator (`op`), and the accumulation operator (if provided) are
2690 tested for domain compatibility as follows:

2691 1. The domain of `Mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.

2692 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.

2693 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.

2694 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.

2695 5. $\mathbf{D}(A)$ and $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.

2696 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
2697 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
2698 the accumulation operator.

2699 Two domains are compatible with each other if values from one domain can be cast to values
2700 in the other domain as per the rules of the C language. In particular, domains from Table 2.2
2701 are all compatible with each other. A domain from a user-defined type is only compatible with
2702 itself. If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain
2703 mismatch error listed above is returned.

2704 From the argument matrices, the internal matrices and mask used in the computation are formed
2705 (\leftarrow denotes copy):

2706 1. Matrix $\tilde{C} \leftarrow C$.

2707 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:

2708 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
2709 $j < \mathbf{ncols}(C)\} \rangle$.

2710 (b) Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) =$
2711 $\text{true}\} \rangle$.

2712 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.

2713 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

2714 4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

2715 The internal matrices and masks are checked for dimension compatibility. The following conditions
2716 must hold:

2717 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{C}})$.

2718 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{C}})$.

2719 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
2720 mismatch error listed above is returned.

2721 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
2722 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2723 We are now ready to carry out the element-wise “sum” and any additional associated operations.
2724 We describe this in terms of two intermediate matrices:

- 2725 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise sum of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 2726 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

2727 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$
2728 is created. The value of each of its elements is computed by

$$2729 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

2730

$$2731 \quad T_{ij} = \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{B}}) \cap \mathbf{ind}(\tilde{\mathbf{A}})))$$

2732

$$2733 \quad T_{ij} = \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{B}}) \cap \mathbf{ind}(\tilde{\mathbf{A}})))$$

2734 where the difference operator in the previous expressions refers to set difference.

2735 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 2736 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 2737 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$2738 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

2739 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
2740 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$2741 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

2742

$$2743 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

2744

$$2745 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

2746

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

2747 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 2748 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 2749 mask which acts as a “write mask”.

- 2750 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 2751 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$2752 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 2753 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 2754 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 2755 mask are unchanged:

$$2756 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

2757 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 2758 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 2759 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 2760 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 2761 sequence.

2762 4.3.6 extract: Selecting Sub-Graphs

2763 Extract a subset of a matrix or vector.

2764 4.3.6.1 extract: Standard vector variant

2765 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector
 2766 whose size is equal to the number of indices.

2767 C Syntax

```
2768     GrB_Info GrB_extract(GrB_Vector      w,
2769                        const GrB_Vector  mask,
2770                        const GrB_BinaryOp accum,
2771                        const GrB_Vector  u,
2772                        const GrB_Index   *indices,
2773                        GrB_Index        nindices,
2774                        const GrB_Descriptor desc);
```

2775 Parameters

2776 w (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 2777 that may be accumulated with the result of the extract operation. On output, this
 2778 vector holds the results of the operation.

2779 **mask** (IN) An optional “write” mask that controls which results from this operation are
 2780 stored into the output vector **w**. The mask dimensions must match those of the
 2781 vector **w** and the domain of the **mask** vector must be of type **bool** or any of the
 2782 predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with
 2783 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

2784 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
 2785 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 2786 specified.

2787 **u** (IN) The GraphBLAS vector from which the subset is extracted.

2788 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of
 2789 elements from **u** that are extracted. If all elements of **u** are to be extracted in order
 2790 from 0 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution
 2791 mode and return value, this array may be manipulated by the caller after this
 2792 operation returns without affecting any deferred computations for this operation.

2793 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

2794 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 2795 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 .

2798 Return Values

2799 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 2800 blocking mode, this indicates that the compatibility tests on di-
 2801 mensions and domains for the input arguments passed successfully.
 2802 Either way, output vector **w** is ready to be used in the next method
 2803 of the sequence.

2804 **GrB_PANIC** Unknown internal error.

2805 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 2806 GraphBLAS objects (input or output) is in an invalid state caused
 2807 by a previous execution error. Call **GrB_error()** to access any error
 2808 messages generated by the implementation.

2809 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

2810 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 2811 a call to **new** (or **dup** for vector parameters).

2812 GrB_INDEX_OUT_OF_BOUNDS A value in indices is greater than or equal to **size(u)**. In non-
2813 blocking mode, this error can be deferred.

2814 GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or nindices \neq **size(w)**.

2815 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each other
2816 or the corresponding domains of the accumulation operator, or the
2817 mask's domain is not compatible with **bool**.

2818 GrB_NULL_POINTER Argument row_indices is a NULL pointer.

2819 Description

2820 This variant of GrB_extract computes the result of extracting a subset of locations from a Graph-
2821 BLAS vector in a specific order: $w = u(\text{indices})$; or, if an optional binary accumulation operator
2822 (\odot) is provided, $w = w \odot u(\text{indices})$. More explicitly:

$$2823 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

2824 Logically, this operation occurs in three steps:

2825 **Setup** The internal vectors and mask used in the computation are formed and their domains
2826 and dimensions are tested for compatibility.

2827 **Compute** The indicated computations are carried out.

2828 **Output** The result is written into the output vector, possibly under control of a mask.

2829 Up to three argument vectors are used in this GrB_extract operation:

- 2830 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 2831 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 2832 3. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

2833 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
2834 bility as follows:

- 2835 1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
- 2836 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(u)$.
- 2837 3. If accum is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
2838 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
2839 mulation operator.

2840 Two domains are compatible with each other if values from one domain can be cast to values in
 2841 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 2842 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 2843 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
 2844 error listed above is returned.

2845 From the arguments, the internal vectors, mask, and index array used in the computation are
 2846 formed (\leftarrow denotes copy):

- 2847 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 2848 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 2849 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 2850 (b) Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 2851 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 2852 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 2853 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument `indices` as follows:
 - 2854 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$.
 - 2855 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$.

2856 The internal vectors and mask are checked for dimension compatibility. The following conditions
 2857 must hold:

- 2858 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 2859 2. $\mathbf{nindices} = \mathbf{size}(\tilde{\mathbf{w}})$.

2860 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
 2861 match error listed above is returned.

2862 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 2863 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2864 We are now ready to carry out the extract and any additional associated operations. We describe
 2865 this in terms of two intermediate vectors:

- 2866 • $\tilde{\mathbf{t}}$: The vector holding the extraction from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.
- 2867 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

2868 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$2869 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, \tilde{\mathbf{u}}(\tilde{\mathbf{I}}[i])) \mid \forall i, 0 \leq i < \mathbf{nindices} : \tilde{\mathbf{I}}[i] \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

2870 At this point, if any value in $\tilde{\mathbf{I}}$ is not in the valid range of indices for vector $\tilde{\mathbf{u}}$, the execution of
 2871 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`
 2872 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
 2873 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

2874 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 2875 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 2876 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$2877 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

2878 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 2879 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$2880 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$2881 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$2882 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

2883 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

2886 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 2887 using what is called a *standard vector mask and replace*. This is carried out under control of the
 2888 mask which acts as a “write mask”.

- 2889 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 2890 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$2891 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 2892 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 2893 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 2894 mask are unchanged:

$$2895 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

2896 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 2897 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 2898 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 2899 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

2900 4.3.6.2 extract: Standard matrix variant

2901 Extract a sub-matrix from a larger matrix as specified by a set of row indices and a set of column
 2902 indices. The result is a matrix whose size is equal to size of the sets of indices.

2903 C Syntax

```
2904         GrB_Info GrB_extract(GrB_Matrix      C,  
2905                             const GrB_Matrix Mask,  
2906                             const GrB_BinaryOp accum,  
2907                             const GrB_Matrix  A,  
2908                             const GrB_Index  *row_indices,  
2909                             GrB_Index      nrows,  
2910                             const GrB_Index  *col_indices,  
2911                             GrB_Index      ncols,  
2912                             const GrB_Descriptor desc);
```

2913 Parameters

2914 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
2915 that may be accumulated with the result of the extract operation. On output, the
2916 matrix holds the results of the operation.

2917 **Mask** (IN) An optional “write” mask that controls which results from this operation are
2918 stored into the output matrix **C**. The mask dimensions must match those of the
2919 matrix **C** and the domain of the **Mask** matrix must be of type **bool** or any of the
2920 predefined “built-in” types in Table 2.2. If the default matrix is desired (i.e., with
2921 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

2922 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
2923 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
2924 specified.

2925 **A** (IN) The GraphBLAS matrix from which the subset is extracted.

2926 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **A**
2927 from which elements are extracted. If elements in all rows of **A** are to be extracted
2928 in order, **GrB_ALL** should be specified. Regardless of execution mode and return
2929 value, this array may be manipulated by the caller after this operation returns
2930 without affecting any deferred computations for this operation.

2931 **nrows** (IN) The number of values in the **row_indices** array. Must be equal to **nrows(C)**.

2932 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns
2933 of **A** from which elements are extracted. If elements in all columns of **A** are to
2934 be extracted in order, then **GrB_ALL** should be specified. Regardless of execution
2935 mode and return value, this array may be manipulated by the caller after this
2936 operation returns without affecting any deferred computations for this operation.

2937 **ncols** (IN) The number of values in the **col_indices** array. Must be equal to **ncols(C)**.

2938 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
2939 should be specified. Non-default field/value pairs are listed as follows:

2940

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.

2941

2942 Return Values

2943

2944

2945

2946

2947

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.

2948

GrB_PANIC Unknown internal error.

2949

2950

2951

2952

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.

2953

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2954

2955

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).

2956

2957

2958

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.

2959

2960

GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, `nrows` \neq `nrows(C)`, or `ncols` \neq `ncols(C)`.

2961

2962

2963

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`.

2964

2965

GrB_NULL_POINTER Either argument `row_indices` is a NULL pointer, argument `col_indices` is a NULL pointer, or both.

2966 Description

2967

2968

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: $C = A(\text{row_indices}, \text{col_indices})$; or, if

2969 an optional binary accumulation operator (\odot) is provided, $C = C \odot A(\text{row_indices}, \text{col_indices})$. More
 2970 explicitly (not accounting for an optional transpose of A):

$$2971 \quad C(i, j) = A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or}$$

$$C(i, j) = C(i, j) \odot A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}$$

2972 Logically, this operation occurs in three steps:

2973 **Setup** The internal matrices and mask used in the computation are formed and their domains
 2974 and dimensions are tested for compatibility.

2975 **Compute** The indicated computations are carried out.

2976 **Output** The result is written into the output matrix, possibly under control of a mask.

2977 Up to three argument matrices are used in the `GrB_extract` operation:

- 2978 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 2979 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \text{nrows}(\text{Mask}), \text{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 2980 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

2981 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
 2982 ibility as follows:

- 2983 1. The domain of `Mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 2984 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 2985 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 2986 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 2987 mulation operator.

2988 Two domains are compatible with each other if values from one domain can be cast to values in
 2989 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 2990 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 2991 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
 2992 error listed above is returned.

2993 From the arguments, the internal matrices, mask, and index arrays used in the computation are
 2994 formed (\leftarrow denotes copy):

- 2995 1. Matrix $\tilde{C} \leftarrow C$.
- 2996 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:
 - 2997 (a) If `Mask = GrB_NULL`, then $\tilde{M} = \langle \text{nrows}(C), \text{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \text{nrows}(C), 0 \leq$
 2998 $j < \text{ncols}(C)\} \rangle$.

- 2999 (b) Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) =$
3000 $\text{true}\} \rangle$.
- 3001 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$.
- 3002 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3003 4. The internal row index array, $\widetilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 3004 (a) If `row_indices = GrB_ALL`, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
- 3005 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 3006 5. The internal column index array, $\widetilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 3007 (a) If `col_indices = GrB_ALL`, then $\widetilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
- 3008 (b) Otherwise, $\widetilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

3009 The internal matrices and mask are checked for dimension compatibility. The following conditions
3010 must hold:

- 3011 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
- 3012 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
- 3013 3. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}$.
- 3014 4. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}$.

3015 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
3016 match error listed above is returned.

3017 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3018 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3019 We are now ready to carry out the extract and any additional associated operations. We describe
3020 this in terms of two intermediate matrices:

- 3021 • $\widetilde{\mathbf{T}}$: The matrix holding the extraction from $\widetilde{\mathbf{A}}$.
- 3022 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3023 The intermediate matrix, $\widetilde{\mathbf{T}}$, is created as follows:

3024
$$\widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}),$$

$$\{(i, j, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \widetilde{\mathbf{J}}[j])) \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (\widetilde{\mathbf{I}}[i], \widetilde{\mathbf{J}}[j]) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

3025 At this point, if any value in the $\widetilde{\mathbf{I}}$ array is not in the range $[0, \mathbf{nrows}(\widetilde{\mathbf{A}}))$ or any value in the $\widetilde{\mathbf{J}}$
3026 array is not in the range $[0, \mathbf{ncols}(\widetilde{\mathbf{A}}))$, the execution of `GrB_extract` ends and the index out-of-
3027 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
3028 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
3029 this point forward in the sequence.

3030 The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3031 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3032 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

3033
$$\tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3034 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3035 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

3036
$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

3037
 3038
$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3039
 3040
$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3041 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3042 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 3043 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 3044 mask which acts as a “write mask”.

- 3045 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 3046 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

3047
$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3048 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 3049 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 3050 mask are unchanged:

3051
$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3052 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3053 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3054 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 3055 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3056 sequence.

3057 4.3.6.3 extract: Column (and row) variant

3058 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the
 3059 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as
 3060 well.

3061 C Syntax

```
3062     GrB_Info GrB_extract(GrB_Vector      w,  
3063                        const GrB_Vector  mask,  
3064                        const GrB_BinaryOp accum,  
3065                        const GrB_Matrix  A,  
3066                        const GrB_Index   *row_indices,  
3067                        GrB_Index        nrows,  
3068                        GrB_Index        col_index,  
3069                        const GrB_Descriptor desc);
```

3070 Parameters

3071 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3072 that may be accumulated with the result of the extract operation. On output, this
3073 vector holds the results of the operation.

3074 **mask** (IN) An optional “write” mask that controls which results from this operation are
3075 stored into the output vector **w**. The mask dimensions must match those of the
3076 vector **w** and the domain of the **mask** vector must be of type **bool** or any of the
3077 predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with
3078 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

3079 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3080 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
3081 specified.

3082 **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

3083 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations
3084 within the specified column of **A** from which elements are extracted. If elements in
3085 all rows of **A** are to be extracted in order, **GrB_ALL** should be specified. Regardless
3086 of execution mode and return value, this array may be manipulated by the caller
3087 after this operation returns without affecting any deferred computations for this
3088 operation.

3089 **nrows** (IN) The number of indices in the **row_indices** array. Must be equal to **size(w)**.

3090 **col_index** (IN) The index of the column of **A** from which to extract values. It must be in the
3091 range $[0, \mathbf{ncols}(A))$.

3092 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
3093 should be specified. Non-default field/value pairs are listed as follows:
3094

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.

3095

3096 Return Values

3097

3098

3099

3100

3101

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.

3102

GrB_PANIC Unknown internal error.

3103

3104

3105

3106

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.

3107

GrB_OUT_OF_MEMORY Not enough memory available for operation.

3108

3109

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).

3110

GrB_INVALID_INDEX **col_index** is outside the allowable range (i.e., greater than **ncols(A)**).

3111

3112

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.

3113

GrB_DIMENSION_MISMATCH **mask** and **w** dimensions are incompatible, or **nrows** \neq **size(w)**.

3114

3115

3116

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**.

3117

GrB_NULL_POINTER Argument **row_indices** is a NULL pointer.

3118 Description

3119

3120

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: $w = A(:, \text{col_index})(\text{row_indices})$; or, if an

3121 optional binary accumulation operator (\odot) is provided, $w = w \odot A(:, \text{col_index})(\text{row_indices})$. More
 3122 explicitly:

$$3123 \quad w(i) = \quad A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or}$$

$$3123 \quad w(i) = w(i) \odot A(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}$$

3124 Logically, this operation occurs in three steps:

3125 **Setup** The internal matrices, vectors, and mask used in the computation are formed and their
 3126 domains and dimensions are tested for compatibility.

3127 **Compute** The indicated computations are carried out.

3128 **Output** The result is written into the output vector, possibly under control of a mask.

3129 Up to three argument vectors and matrices are used in this `GrB_extract` operation:

- 3130 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3131 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3132 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

3133 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain
 3134 compatibility as follows:

- 3135 1. The domain of `mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 3136 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(A)$.
- 3137 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3138 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 3139 mulation operator.

3140 Two domains are compatible with each other if values from one domain can be cast to values in
 3141 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 3142 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3143 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
 3144 error listed above is returned.

3145 From the arguments, the internal vector, matrix, mask, and index array used in the computation
 3146 are formed (\leftarrow denotes copy):

- 3147 1. Vector $\tilde{w} \leftarrow w$.
- 3148 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
 - 3149 (a) If `mask = GrB_NULL`, then $\tilde{m} = \langle \mathbf{size}(w), \{i, \forall i : 0 \leq i < \mathbf{size}(w)\} \rangle$.
 - 3150 (b) Otherwise, $\tilde{m} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

3151 (c) If desc[GrB_MASK].GrB_SCMP is set, then $\tilde{\mathbf{m}} \leftarrow -\tilde{\mathbf{m}}$.

3152 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

3153 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument row_indices as follows:

3154 (a) If indices = GrB_ALL, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.

3155 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$.

3156 The internal vector, mask, and index array are checked for dimension compatibility. The following
3157 conditions must hold:

3158 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

3159 2. $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}$.

3160 If any compatibility rule above is violated, execution of GrB_extract ends and the dimension mis-
3161 match error listed above is returned.

3162 The col_index parameter is checked for a valid value. The following condition must hold:

3163 1. $0 \leq \text{col_index} < \text{ncols}(\mathbf{A})$

3164 If the rule above is violated, execution of GrB_extract ends and the invalid index error listed above
3165 is returned.

3166 From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with
3167 GrB_SUCCESS return code and defer any computation and/or execution error codes.

3168 We are now ready to carry out the extract and any additional associated operations. We describe
3169 this in terms of two intermediate vectors:

3170 • $\tilde{\mathbf{t}}$: The vector holding the extraction from a column of $\tilde{\mathbf{A}}$.

3171 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3172 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

3173
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \text{col_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\tilde{\mathbf{I}}[i], \text{col_index}) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

3174 At this point, if any value in $\tilde{\mathbf{I}}$ is not in the range $[0, \text{nrows}(\tilde{\mathbf{A}}))$, the execution of GrB_extract
3175 ends and the index-out-of-bounds error listed above is generated. In GrB_NONBLOCKING mode,
3176 the error can be deferred until a sequence-terminating GrB_wait() is called. Regardless, the result
3177 vector, \mathbf{w} , is invalid from this point forward in the sequence.

3178 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

3179 • If accum = GrB_NULL, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3180 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3181 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3182 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3183 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3184 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

3185

$$3186 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3187

$$3188 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3189 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3190 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3191 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3192 mask which acts as a “write mask”.

3193 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 3194 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3195 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3196 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3197 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3198 mask are unchanged:

$$3199 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3200 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 3201 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 3202 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 3203 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

3204 4.3.7 assign: Modifying Sub-Graphs

3205 Assign the contents of a subset of a matrix or vector.

3206 4.3.7.1 assign: Standard vector variant

3207 Assign values (and implied zeros) from one GraphBLAS vector to a subset of a vector as specified
 3208 by a set of indices. The size of the input vector is the same size as the index array provided.

3209 **C Syntax**

```

3210     GrB_Info GrB_assign(GrB_Vector      w,
3211                        const GrB_Vector  mask,
3212                        const GrB_BinaryOp accum,
3213                        const GrB_Vector  u,
3214                        const GrB_Index   *indices,
3215                        GrB_Index        nindices,
3216                        const GrB_Descriptor desc);

```

3217 **Parameters**

3218 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3219 that may be accumulated with the result of the assign operation. On output, this
3220 vector holds the results of the operation.

3221 **mask** (IN) An optional “write” mask that controls which results from this operation are
3222 stored into the output vector **w**. The mask dimensions must match those of the
3223 vector **w** and the domain of the **mask** vector must be of type **bool** or any of the
3224 predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with
3225 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

3226 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3227 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
3228 specified.

3229 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

3230 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
3231 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0
3232 to **nindices** – 1, then **GrB_ALL** should be specified. Regardless of execution mode
3233 and return value, this array may be manipulated by the caller after this operation
3234 returns without affecting any deferred computations for this operation. If this
3235 array contains duplicate values, it implies in assignment of more than one value to
3236 the same location which leads to undefined results.

3237 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

3238 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
3239 should be specified. Non-default field/value pairs are listed as follows:

3240

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 .

3241

3242 Return Values

3243 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3244 blocking mode, this indicates that the compatibility tests on di-
3245 mensions and domains for the input arguments passed successfully.
3246 Either way, output vector w is ready to be used in the next method
3247 of the sequence.

3248 GrB_PANIC Unknown internal error.

3249 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3250 GraphBLAS objects (input or output) is in an invalid state caused
3251 by a previous execution error. Call `GrB_error()` to access any error
3252 messages generated by the implementation.

3253 GrB_OUT_OF_MEMORY Not enough memory available for operation.

3254 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3255 a call to `new` (or `dup` for vector parameters).

3256 GrB_INDEX_OUT_OF_BOUNDS A value in indices is greater than or equal to `size(w)`. In non-
3257 blocking mode, this can be reported as an execution error.

3258 GrB_DIMENSION_MISMATCH `mask` and `w` dimensions are incompatible, or `nindices` \neq `size(u)`.

3259 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each other
3260 or the corresponding domains of the accumulation operator, or the
3261 mask's domain is not compatible with `bool`.

3262 GrB_NULL_POINTER Argument `indices` is a NULL pointer.

3263 Description

3264 This variant of `GrB_assign` computes the result of assigning elements from a source GraphBLAS
3265 vector to a destination GraphBLAS vector in a specific order: $w(\text{indices}) = u$; or, if an optional
3266 binary accumulation operator (\odot) is provided, $w(\text{indices}) = w(\text{indices}) \odot u$. More explicitly:

$$\begin{aligned} 3267 \quad w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < n\text{indices}, \text{ or} \\ w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < n\text{indices}. \end{aligned}$$

3268 Logically, this operation occurs in three steps:

3269 **Setup** The internal vectors and mask used in the computation are formed and their domains
3270 and dimensions are tested for compatibility.

3271 **Compute** The indicated computations are carried out.

3272 **Output** The result is written into the output vector, possibly under control of a mask.

3273 Up to three argument vectors are used in the `GrB_assign` operation:

- 3274 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3275 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3276 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

3277 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
3278 bility as follows:

- 3279 1. The domain of `mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 3280 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 3281 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
3282 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
3283 mulation operator.

3284 Two domains are compatible with each other if values from one domain can be cast to values in
3285 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
3286 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3287 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
3288 error listed above is returned.

3289 From the arguments, the internal vectors, `mask` and index array used in the computation are formed
3290 (\leftarrow denotes copy):

- 3291 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3292 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3293 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3294 (b) Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 3295 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3296 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 3297 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument `indices` as follows:
 - 3298 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$.
 - 3299 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$.

3300 The internal vector and `mask` are checked for dimension compatibility. The following conditions
3301 must hold:

- 3302 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$

3303 2. $nindices = \mathbf{size}(\tilde{\mathbf{u}})$.

3304 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
3305 error listed above is returned.

3306 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3307 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3308 We are now ready to carry out the assign and any additional associated operations. We describe
3309 this in terms of two intermediate vectors:

- 3310 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.
- 3311 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3312 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$3313 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < nindices : i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

3314 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{w}}$, computation
3315 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`
3316 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
3317 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

3318 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 3319 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$3320 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3321 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
3322 of $\tilde{\mathbf{w}}$ ($\mathbf{ind}(\tilde{\mathbf{w}})$) and remove from it all the indices of $\tilde{\mathbf{w}}$ that are in the set of indices being
3323 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

3324 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3325 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3326 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$3327 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

3329 where the difference operator refers to set difference.

- 3330 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3331 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3332 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3333 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3334 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$3335 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$3336 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3339 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3340 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3341 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3342 mask which acts as a “write mask”.

- 3343 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 3344 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3345 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3346 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3347 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3348 mask are unchanged:

$$3349 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3350 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 3351 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 3352 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 3353 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

3354 4.3.7.2 assign: Standard matrix variant

3355 Assign values (and implied zeros) from one GraphBLAS matrix to a subset of a matrix as specified
 3356 by a set of indices. The dimensions of the input matrix are the same size as the row and column
 3357 index arrays provided.

3358 C Syntax

```
3359     GrB_Info GrB_assign(GrB_Matrix      C,
3360                       const GrB_Matrix Mask,
3361                       const GrB_BinaryOp accum,
3362                       const GrB_Matrix A,
3363                       const GrB_Index *row_indices,
3364                       GrB_Index      nrows,
3365                       const GrB_Index *col_indices,
3366                       GrB_Index      ncols,
3367                       const GrB_Descriptor desc);
```

3368 Parameters

3369 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 3370 that may be accumulated with the result of the assign operation. On output, the
 3371 matrix holds the results of the operation.

3372 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 3373 stored into the output matrix **C**. The mask dimensions must match those of the
 3374 matrix **C** and the domain of the **Mask** matrix must be of type **bool** or any of the
 3375 predefined “built-in” types in Table 2.2. If the default matrix is desired (i.e., with
 3376 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

3377 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
 3378 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 3379 specified.

3380 **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

3381 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
 3382 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** - 1,
 3383 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
 3384 this array may be manipulated by the caller after this operation returns without
 3385 affecting any deferred computations for this operation. If this array contains du-
 3386 plicate values, it implies assignment of more than one value to the same location
 3387 which leads to undefined results.

3388 **nrows** (IN) The number of values in the **row_indices** array. Must be equal to **nrows(A)** if
 3389 **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

3390 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns
 3391 of **C** that are assigned. If all columns of **C** are to be assigned in order from 0 to
 3392 **ncols** - 1, then **GrB_ALL** should be specified. Regardless of execution mode and
 3393 return value, this array may be manipulated by the caller after this operation
 3394 returns without affecting any deferred computations for this operation. If this
 3395 array contains duplicate values, it implies assignment of more than one value to
 3396 the same location which leads to undefined results.

3397 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **ncols(A)** if **A** is
 3398 not transposed, or equal to **nrows(A)** if **A** is transposed.

3399 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 3400 should be specified. Non-default field/value pairs are listed as follows:

3401

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.

3402

3403 Return Values

3404 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3405 blocking mode, this indicates that the compatibility tests on di-
3406 mensions and domains for the input arguments passed successfully.
3407 Either way, output matrix C is ready to be used in the next method
3408 of the sequence.

3409 GrB_PANIC Unknown internal error.

3410 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3411 GraphBLAS objects (input or output) is in an invalid state caused
3412 by a previous execution error. Call `GrB_error()` to access any error
3413 messages generated by the implementation.

3414 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3415 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3416 a call to `new` (or `Matrix_dup` for matrix parameters).

3417 GrB_INDEX_OUT_OF_BOUNDS A value in `row_indices` is greater than or equal to `nrows(C)`, or a
3418 value in `col_indices` is greater than or equal to `ncols(C)`. In non-
3419 blocking mode, this can be reported as an execution error.

3420 GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, `nrows` \neq `nrows(A)`, or
3421 `ncols` \neq `ncols(A)`.

3422 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with each
3423 other or the corresponding domains of the accumulation operator,
3424 or the mask's domain is not compatible with `bool`.

3425 GrB_NULL_POINTER Either argument `row_indices` is a NULL pointer, argument `col_indices`
3426 is a NULL pointer, or both.

3427 Description

3428 This variant of `GrB_assign` computes the result of assigning the contents of A to a subset of rows
3429 and columns in C in a specified order: $C(\text{row_indices}, \text{col_indices}) = A$; or, if an optional binary
3430 accumulation operator (\odot) is provided, $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot A$.
3431 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 3432 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot A(i, j), \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

3433 Logically, this operation occurs in three steps:

3434 Setup The internal matrices and mask used in the computation are formed and their domains
3435 and dimensions are tested for compatibility.

3436 Compute The indicated computations are carried out.

3437 Output The result is written into the output matrix, possibly under control of a mask.

3438 Up to three argument matrices are used in the `GrB_assign` operation:

3439 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

3440 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)

3441 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

3442 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
3443 ibility as follows:

3444 1. The domain of `Mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.

3445 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.

3446 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3447 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
3448 mulation operator.

3449 Two domains are compatible with each other if values from one domain can be cast to values in
3450 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
3451 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3452 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
3453 error listed above is returned.

3454 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are
3455 formed (\leftarrow denotes copy):

3456 1. Matrix $\tilde{C} \leftarrow C$.

3457 2. Two-dimensional mask \tilde{M} is computed from argument `Mask` as follows:

3458 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
3459 $j < \mathbf{ncols}(C)\} \rangle$.

3460 (b) Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) =$
3461 $\text{true}\} \rangle$.

3462 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.

3463 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

3464 4. The internal row index array, \tilde{I} , is computed from argument `row_indices` as follows:

3465 (a) If `row_indices` = `GrB_ALL`, then $\tilde{I}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.

3466 (b) Otherwise, $\tilde{I}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.

3467 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:

3468 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$.

3469 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \text{ncols}$.

3470 The internal matrices and mask are checked for dimension compatibility. The following conditions
3471 must hold:

3472 1. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$.

3473 2. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$.

3474 3. $\text{nrows}(\tilde{\mathbf{A}}) = \text{nrows}$.

3475 4. $\text{ncols}(\tilde{\mathbf{A}}) = \text{ncols}$.

3476 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
3477 error listed above is returned.

3478 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3479 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3480 We are now ready to carry out the assign and any additional associated operations. We describe
3481 this in terms of two intermediate vectors:

3482 • $\tilde{\mathbf{T}}$: The matrix holding the contents from $\tilde{\mathbf{A}}$ in their destination locations relative to $\tilde{\mathbf{C}}$.

3483 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3484 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$3485 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{T}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (i, j) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

3486 At this point, if any value in the $\tilde{\mathbf{I}}$ array is not in the range $[0, \text{nrows}(\tilde{\mathbf{C}}))$ or any value in the
3487 $\tilde{\mathbf{J}}$ array is not in the range $[0, \text{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-
3488 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
3489 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
3490 this point forward in the sequence.

3491 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

3492 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$3493 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \\ 3494 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\text{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{T}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \text{ind}(\tilde{\mathbf{C}}))) \cup \text{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3495 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 3496 of $\tilde{\mathbf{C}}$ ($\mathbf{ind}(\tilde{\mathbf{C}})$) and remove from it all the indices of $\tilde{\mathbf{C}}$ that are in the set of indices being
 3497 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

3498 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3499 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3500 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3501

$$3502 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

3503 where the difference operator refers to set difference.

- 3504 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3505 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3506 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3507 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3508 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

3509

$$3510 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3511

$$3512 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3513 where $\odot = \bigcirc(\text{accum})$, and the difference operator refers to set difference.

3514 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 3515 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 3516 mask which acts as a “write mask”.

- 3517 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 3518 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3519 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3520 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 3521 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 3522 mask are unchanged:

$$3523 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3524 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3525 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3526 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 3527 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3528 sequence.

3529 **4.3.7.3 assign: Column variant**

3530 Assign the contents a vector to a subset of elements in one column of a matrix. Note that since
3531 the output cannot be transposed, a different variant of `assign` is provided to assign to a row of a
3532 matrix.

3533 **C Syntax**

```
3534         GrB_Info GrB_assign(GrB_Matrix          C,  
3535                             const GrB_Vector   mask,  
3536                             const GrB_BinaryOp accum,  
3537                             const GrB_Vector   u,  
3538                             const GrB_Index   *row_indices,  
3539                             GrB_Index        nrows,  
3540                             GrB_Index        col_index,  
3541                             const GrB_Descriptor desc);
```

3542 **Parameters**

3543 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3544 that may be accumulated with the result of the assign operation. On output, this
3545 matrix holds the results of the operation.

3546 **mask** (IN) An optional “write” mask that controls which results from this operation are
3547 stored into the specified column of the output matrix `C`. The mask dimensions
3548 must match those of a single column of the matrix `C` and the domain of the Mask
3549 matrix must be of type `bool` or any of the predefined “built-in” types in Table 2.2.
3550 If the default vector is desired (i.e., with correct dimensions and filled with `true`),
3551 `GrB_NULL` should be specified.

3552 **accum** (IN) An optional binary operator used for accumulating entries into existing `C`
3553 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3554 specified.

3555 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column
3556 of `C`.

3557 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
3558 the specified column of `C` that are to be assigned. If all elements of the column
3559 in `C` are to be assigned in order from index 0 to `nrows - 1`, then `GrB_ALL` should
3560 be specified. Regardless of execution mode and return value, this array may be
3561 manipulated by the caller after this operation returns without affecting any de-
3562 ferred computations for this operation. If this array contains duplicate values, it
3563 implies in assignment of more than one value to the same location which leads to
3564 undefined results.

3565 `nrows` (IN) The number of values in `row_indices` array. Must be equal to `size(u)`.
3566 `col_index` (IN) The index of the column in `C` to assign. Must be in the range `[0, ncols(C))`.
3567 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
3568 should be specified. Non-default field/value pairs are listed as follows:
3569

Param	Field	Value	Description
<code>C</code>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output column in <code>C</code> is cleared (all elements removed) before result is stored in it.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_SCMP</code>	Use the structural complement of <code>mask</code> .

3570

3571 Return Values

3572 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
3573 blocking mode, this indicates that the compatibility tests on di-
3574 mensions and domains for the input arguments passed successfully.
3575 Either way, output matrix `C` is ready to be used in the next method
3576 of the sequence.

3577 `GrB_PANIC` Unknown internal error.

3578 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
3579 GraphBLAS objects (input or output) is in an invalid state caused
3580 by a previous execution error. Call `GrB_error()` to access any error
3581 messages generated by the implementation.

3582 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

3583 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by
3584 a call to `new` (or `dup` for vector or matrix parameters).

3585 `GrB_INVALID_INDEX` `col_index` is outside the allowable range (i.e., greater than `ncols(C)`).

3586 `GrB_INDEX_OUT_OF_BOUNDS` A value in `row_indices` is greater than or equal to `nrows(C)`. In
3587 non-blocking mode, this can be reported as an execution error.

3588 `GrB_DIMENSION_MISMATCH` `mask` size and number of rows in `C` are not the same, or `nrows` \neq
3589 `size(u)`.

3590 `GrB_DOMAIN_MISMATCH` The domains of the matrix and vector are incompatible with each
3591 other or the corresponding domains of the accumulation operator,
3592 or the mask's domain is not compatible with `bool`.

3593 `GrB_NULL_POINTER` Argument `row_indices` is a NULL pointer.

3594 **Description**

3595 This variant of `GrB_assign` computes the result of assigning a subset of locations in a column of a
3596 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

3597 $C(:, \text{col_index}) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(:, \text{col_index}) =$
3598 $C(:, \text{col_index}) \odot u$. Taking order of `row_indices` into account, it is more explicitly written as:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_index}) = u(i), \quad \forall i : 0 \leq i < \text{nrows}, \text{ or} \\ & C(\text{row_indices}[i], \text{col_index}) = C(\text{row_indices}[i], \text{col_index}) \odot u(i), \quad \forall i : 0 \leq i < \text{nrows}. \end{aligned}$$

3600 Logically, this operation occurs in three steps:

3601 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
3602 domains and dimensions are tested for compatibility.

3603 **Compute** The indicated computations are carried out.

3604 **Output** The result is written into the output matrix, possibly under control of a mask.

3605 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 3606 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3607 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3608 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3609 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
3610 compatibility as follows:

- 3611 1. The domain of `mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 3612 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(u)$.
- 3613 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3614 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
3615 mulation operator.

3616 Two domains are compatible with each other if values from one domain can be cast to values in
3617 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
3618 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3619 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
3620 error listed above is returned.

3621 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 3622 1. $0 \leq \text{col_index} < \mathbf{ncols}(C)$

3623 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
 3624 is returned.

3625 From the arguments, the internal vectors, mask, and index array used in the computation are
 3626 formed (\leftarrow denotes copy):

3627 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a column of \mathbf{C} as follows:

$$3628 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

3629 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

- 3630 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$.
 3631 (b) Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
 3632 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

3633 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3634 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:

- 3635 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
 3636 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.

3637 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
 3638 conditions must hold:

- 3639 1. $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$
 3640 2. $\mathbf{nrows} = \mathbf{size}(\tilde{\mathbf{u}})$.

3641 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
 3642 error listed above is returned.

3643 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3644 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3645 We are now ready to carry out the assign and any additional associated operations. We describe
 3646 this in terms of two intermediate vectors:

- 3647 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 3648 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3649 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$3650 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \mathbf{nrows} : i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

3651 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
 3652 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`

3653 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
 3654 result matrix, C , is invalid from this point forward in the sequence.

3655 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 3656 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$3657 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(C), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3658 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 3659 of $\tilde{\mathbf{c}}$ ($\mathbf{ind}(\tilde{\mathbf{c}})$) and remove from it all the indices of $\tilde{\mathbf{c}}$ that are in the set of indices being
 3660 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

3661 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3662 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$3663 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$3664 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

3666 where the difference operator refers to set difference.

- 3667 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3668 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i) \forall i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3669 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3670 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3671 \quad z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$3672 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$3673 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

3676 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3677 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
 3678 result matrix, $C(:, \text{col_index})$. This is carried out under control of the mask which acts as a “write
 3679 mask”.

- 3680 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $C(:, \text{col_index})$ on input to this
 3681 operation are deleted and the new contents of the column is given by:

$$3682 \quad \mathbf{L}(C) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3683 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3684 copied into the column of the final result matrix, $C(:, \text{col_index})$, and elements of this column
 3685 that fall outside the set indicated by the mask are unchanged:

$$3686 \quad \mathbf{L}(C) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup$$

$$3687 \quad \{(i, \text{col_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$3688 \quad \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3689 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
3690 of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
3691 exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may
3692 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

3693 4.3.7.4 assign: Row variant

3694 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the
3695 output cannot be transposed, a different variant of assign is provided to assign to a column of a
3696 matrix.

3697 C Syntax

```
3698         GrB_Info GrB_assign(GrB_Matrix      C,  
3699                          const GrB_Vector  mask,  
3700                          const GrB_BinaryOp accum,  
3701                          const GrB_Vector  u,  
3702                          GrB_Index        row_index,  
3703                          const GrB_Index  *col_indices,  
3704                          GrB_Index        ncols,  
3705                          const GrB_Descriptor desc);
```

3706 Parameters

3707 C (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values
3708 that may be accumulated with the result of the assign operation. On output, this
3709 matrix holds the results of the operation.

3710 mask (IN) An optional “write” mask that controls which results from this operation are
3711 stored into the specified row of the output matrix C. The mask dimensions must
3712 match those of a single row of the matrix C and the domain of the Mask matrix
3713 must be of type bool or any of the predefined “built-in” types in Table 2.2. If
3714 the default vector is desired (i.e., with correct dimensions and filled with true),
3715 GrB_NULL should be specified.

3716 accum (IN) An optional binary operator used for accumulating entries into existing C
3717 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3718 specified.

3719 u (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of
3720 C.

3721 row_index (IN) The index of the row in C to assign. Must be in the range [0, nrows(C)].

3722 col_indices (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
3723 the specified row of C that are to be assigned. If all elements of the row in C are to

3724 be assigned in order from index 0 to `ncols - 1`, then `GrB_ALL` should be specified.
 3725 Regardless of execution mode and return value, this array may be manipulated by
 3726 the caller after this operation returns without affecting any deferred computations
 3727 for this operation. If this array contains duplicate values, it implies in assignment
 3728 of more than one value to the same location which leads to undefined results.

3729 `ncols` (IN) The number of values in `col_indices` array. Must be equal to `size(u)`.

3730 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 3731 should be specified. Non-default field/value pairs are listed as follows:
 3732

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 <code>mask</code> .

3734 Return Values

3735 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 3736 blocking mode, this indicates that the compatibility tests on di-
 3737 mensions and domains for the input arguments passed successfully.
 3738 Either way, output matrix C is ready to be used in the next method
 3739 of the sequence.

3740 `GrB_PANIC` Unknown internal error.

3741 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 3742 GraphBLAS objects (input or output) is in an invalid state caused
 3743 by a previous execution error. Call `GrB_error()` to access any error
 3744 messages generated by the implementation.

3745 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

3746 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by
 3747 a call to `new` (or `dup` for vector or matrix parameters).

3748 `GrB_INVALID_INDEX` `row_index` is outside the allowable range (i.e., greater than `nrows(C)`).

3749 `GrB_INDEX_OUT_OF_BOUNDS` A value in `col_indices` is greater than or equal to `ncols(C)`. In non-
 3750 blocking mode, this can be reported as an execution error.

3751 `GrB_DIMENSION_MISMATCH` `mask` size and number of columns in C are not the same, or `ncols` \neq
 3752 `size(u)`.

3753 `GrB_DOMAIN_MISMATCH` The domains of the matrix and vector are incompatible with each
 3754 other or the corresponding domains of the accumulation operator,
 3755 or the mask's domain is not compatible with `bool`.

3756

GrB_NULL_POINTER Argument col_indices is a NULL pointer.

3757 **Description**

3758 This variant of GrB_assign computes the result of assigning a subset of locations in a row of a
3759 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

3760 $C(\text{row_index}, :) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(\text{row_index}, :$
3761 $) = C(\text{row_index}, :) \odot u$. Taking order of col_indices into account it is more explicitly written as:

$$3762 \quad C(\text{row_index}, \text{col_indices}[j]) = u(j), \quad \forall j : 0 \leq j < \text{ncols}, \text{ or}$$

$$C(\text{row_index}, \text{col_indices}[j]) = C(\text{row_index}, \text{col_indices}[j]) \odot u(j), \quad \forall j : 0 \leq j < \text{ncols}$$

3763 Logically, this operation occurs in three steps:

3764 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
3765 domains and dimensions are tested for compatibility.

3766 **Compute** The indicated computations are carried out.

3767 **Output** The result is written into the output matrix, possibly under control of a mask.

3768 Up to three argument vectors and matrices are used in this GrB_assign operation:

- 3769 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3770 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3771 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3772 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
3773 compatibility as follows:

- 3774 1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
- 3775 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(u)$.
- 3776 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3777 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
3778 mulation operator.

3779 Two domains are compatible with each other if values from one domain can be cast to values in
3780 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
3781 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3782 any compatibility rule above is violated, execution of GrB_assign ends and the domain mismatch
3783 error listed above is returned.

3784 The row_index parameter is checked for a valid value. The following condition must hold:

3785 1. $0 \leq \text{row_index} < \text{nrows}(\mathbf{C})$

3786 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
3787 is returned.

3788 From the arguments, the internal vectors, `mask`, and index array used in the computation are
3789 formed (\leftarrow denotes copy):

3790 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a row of \mathbf{C} as follows:

3791
$$\tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \text{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \text{ncols}(\mathbf{C}), i = \text{row_index}, (i, j) \in \text{ind}(\mathbf{C})\} \rangle$$

3792 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

3793 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \text{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \text{ncols}(\mathbf{C})\} \rangle$.

3794 (b) Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

3795 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

3796 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3797 4. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:

3798 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$.

3799 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \text{ncols}$.

3800 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
3801 conditions must hold:

3802 1. $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$

3803 2. $\text{ncols} = \text{size}(\tilde{\mathbf{u}})$.

3804 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
3805 error listed above is returned.

3806 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3807 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3808 We are now ready to carry out the assign and any additional associated operations. We describe
3809 this in terms of two intermediate vectors:

3810 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.

3811 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3812 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

3813
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \text{ncols} : j \in \text{ind}(\tilde{\mathbf{u}})\} \rangle$$

3814 At this point, if any value of $\tilde{\mathbf{J}}[j]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
3815 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
3816 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
3817 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

3818 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 3819 • If `accum = GrB.NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$3820 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3821 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
3822 of $\tilde{\mathbf{c}}$ ($\mathbf{ind}(\tilde{\mathbf{c}})$) and remove from it all the indices of $\tilde{\mathbf{c}}$ that are in the set of indices being
3823 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

3824 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3825 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$3826 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$3827 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

3828 where the difference operator refers to set difference.

- 3830 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3831 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3832 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3833 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3834 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$3835 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$3836 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

3837 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3840 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
3841 result matrix, $\mathbf{C}(\text{row_index}, :)$. This is carried out under control of the mask which acts as a “write
3842 mask”.

- 3843 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(\text{row_index}, :)$ on input to this
3844 operation are deleted and the new contents of the column is given by:

$$3845 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3846 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
3847 copied into the column of the final result matrix, $\mathbf{C}(\text{row_index}, :)$, and elements of this column
3848 that fall outside the set indicated by the mask are unchanged:

$$3849 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup$$

$$3850 \quad \{(\text{row_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$3851 \quad \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3852 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
3853 of vector w is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
3854 exits with return value GrB_SUCCESS and the new content of vector w is as defined above but may
3855 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

3856 4.3.7.5 assign: Constant vector variant

3857 Assign the same value to a specified subset of vector elements. With the use of GrB_ALL, the entire
3858 destination vector can be filled with the constant.

3859 C Syntax

```
3860         GrB_Info GrB_assign(GrB_Vector      w,  
3861                           const GrB_Vector  mask,  
3862                           const GrB_BinaryOp accum,  
3863                           <type>          val,  
3864                           const GrB_Index   *indices,  
3865                           GrB_Index        nindices,  
3866                           const GrB_Descriptor desc);
```

3867 Parameters

3868 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3869 that may be accumulated with the result of the assign operation. On output, this
3870 vector holds the results of the operation.

3871 **mask** (IN) An optional “write” mask that controls which results from this operation are
3872 stored into the output vector w. The mask dimensions must match those of the
3873 vector w and the domain of the mask vector must be of type bool or any of the
3874 predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with
3875 correct dimensions and filled with true), GrB_NULL should be specified.

3876 **accum** (IN) An optional binary operator used for accumulating entries into existing w
3877 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3878 specified.

3879 **val** (IN) Scalar value to assign to (a subset of) w.

3880 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
3881 w that are to be assigned. If all elements of w are to be assigned in order from 0
3882 to nindices - 1, then GrB_ALL should be specified. Regardless of execution mode
3883 and return value, this array may be manipulated by the caller after this operation
3884 returns without affecting any deferred computations for this operation. In this
3885 variant, the specific order of the values in the array has no effect on the result.
3886 Unlike other variants, if there are duplicated values in this array the result is still
3887 defined.

3888 nindices (IN) The number of values in indices array. Must be in the range: $[0, \text{size}(w)]$. If
3889 nindices is zero, the operation becomes a NO-OP.

3890 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
3891 should be specified. Non-default field/value pairs are listed as follows:
3892

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.

3894 Return Values

3895 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3896 blocking mode, this indicates that the compatibility tests on di-
3897 mensions and domains for the input arguments passed successfully.
3898 Either way, output vector w is ready to be used in the next method
3899 of the sequence.

3900 GrB_PANIC Unknown internal error.

3901 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3902 GraphBLAS objects (input or output) is in an invalid state caused
3903 by a previous execution error. Call GrB_error() to access any error
3904 messages generated by the implementation.

3905 GrB_OUT_OF_MEMORY Not enough memory available for operation.

3906 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3907 a call to new (or dup for vector parameters).

3908 GrB_INDEX_OUT_OF_BOUNDS A value in indices is greater than or equal to **size(w)**. In non-
3909 blocking mode, this can be reported as an execution error.

3910 GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or nindices is not less than
3911 **size(w)**.

3912 GrB_DOMAIN_MISMATCH The domains of the vector and scalar are incompatible with each
3913 other or the corresponding domains of the accumulation operator,
3914 or the mask's domain is not compatible with bool.

3915 GrB_NULL_POINTER Argument indices is a NULL pointer.

3916 **Description**

3917 This variant of `GrB_assign` computes the result of assigning a constant scalar value to locations in
3918 a destination GraphBLAS vector: $w(\text{indices}) = \text{val}$; or, if an optional binary accumulation operator
3919 (\odot) is provided, $w(\text{indices}) = w(\text{indices}) \odot \text{val}$. More explicitly:

$$\begin{aligned} 3920 \quad & w(\text{indices}[i]) = \text{val}, \quad \forall i : 0 \leq i < \text{nindices}, \quad \text{or} \\ & w(\text{indices}[i]) = w(\text{indices}[i]) \odot \text{val}, \quad \forall i : 0 \leq i < \text{nindices}. \end{aligned}$$

3921 Logically, this operation occurs in three steps:

3922 **Setup** The internal vectors and mask used in the computation are formed and their domains
3923 and dimensions are tested for compatibility.

3924 **Compute** The indicated computations are carried out.

3925 **Output** The result is written into the output vector, possibly under control of a mask.

3926 Up to two argument vectors are used in the `GrB_assign` operation:

- 3927 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3928 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

3929 The argument scalar, vectors, and the accumulation operator (if provided) are tested for domain
3930 compatibility as follows:

- 3931 1. The domain of `mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 3932 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(\text{val})$.
- 3933 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3934 of the accumulation operator and $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
3935 mulation operator.

3936 Two domains are compatible with each other if values from one domain can be cast to values in
3937 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
3938 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3939 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
3940 error listed above is returned.

3941 From the arguments, the internal vectors, mask and index array used in the computation are formed
3942 (\leftarrow denotes copy):

- 3943 1. Vector $\tilde{w} \leftarrow w$.
- 3944 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:

- 3945 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 3946 (b) Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 3947 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 3948 3. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:
- 3949 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$.
- 3950 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$.

3951 The internal vector and mask are checked for dimension compatibility. The following conditions
3952 must hold:

- 3953 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 3954 2. $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$.

3955 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
3956 error listed above is returned.

3957 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3958 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3959 We are now ready to carry out the assign and any additional associated operations. We describe
3960 this in terms of two intermediate vectors:

- 3961 • $\tilde{\mathbf{t}}$: The vector holding the copies of the scalar `val` in their destination locations relative to $\tilde{\mathbf{w}}$.
- 3962 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3963 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$3964 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

3965 If $\tilde{\mathbf{I}}$ is empty, this operation results in an empty vector, $\tilde{\mathbf{t}}$. Otherwise, if any value in the $\tilde{\mathbf{I}}$ array
3966 is not in the range $[0, \mathbf{size}(\tilde{\mathbf{w}}))$, the execution of `GrB_assign` ends and the index out-of-bounds
3967 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a
3968 sequence-terminating `GrB_wait()` is called. Regardless, the result vector, \mathbf{w} , is invalid from this
3969 point forward in the sequence.

3970 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 3971 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$3972 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3973 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
3974 of $\tilde{\mathbf{w}}$ ($\mathbf{ind}(\tilde{\mathbf{w}})$) and remove from it all the indices of $\tilde{\mathbf{w}}$ that are in the set of indices being
3975 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

3976 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3977 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3978 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$3979 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

3981 where the difference operator refers to set difference. We note that in this case of assigning
 3982 a constant, $\{\tilde{\mathbf{I}}[k], \forall k\}$ and $\mathbf{ind}(\tilde{\mathbf{t}})$ are identical.

- 3983 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3984 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3985 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3986 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3987 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$3988 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$3989 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3992 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3993 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3994 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3995 mask which acts as a “write mask”.

- 3996 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 3997 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3998 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3999 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 4000 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 4001 mask are unchanged:

$$4002 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4003 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 4004 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 4005 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 4006 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4007 4.3.7.6 assign: Constant matrix variant

4008 Assign the same value to a specified subset of matrix elements. With the use of `GrB_ALL`, the entire
 4009 destination matrix can be filled with the constant.

4010 C Syntax

```
4011     GrB_Info GrB_assign(GrB_Matrix      C,  
4012                       const GrB_Matrix Mask,  
4013                       const GrB_BinaryOp accum,  
4014                       <type>          val,  
4015                       const GrB_Index  *row_indices,  
4016                       GrB_Index       nrows,  
4017                       const GrB_Index  *col_indices,  
4018                       GrB_Index       ncols,  
4019                       const GrB_Descriptor desc);
```

4020 Parameters

4021 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
4022 that may be accumulated with the result of the assign operation. On output, the
4023 matrix holds the results of the operation.

4024 **Mask** (IN) An optional “write” mask that controls which results from this operation are
4025 stored into the output matrix **C**. The mask dimensions must match those of the
4026 matrix **C** and the domain of the **Mask** matrix must be of type **bool** or any of the
4027 predefined “built-in” types in Table 2.2. If the default matrix is desired (i.e., with
4028 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

4029 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4030 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4031 specified.

4032 **val** (IN) Scalar value to assign to (a subset of) **C**.

4033 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**
4034 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** - 1,
4035 then **GrB_ALL** can be specified. Regardless of execution mode and return value,
4036 this array may be manipulated by the caller after this operation returns without
4037 affecting any deferred computations for this operation. Unlike other variants, if
4038 there are duplicated values in this array the result is still defined.

4039 **nrows** (IN) The number of values in **row_indices** array. Must be in the range: [0, **nrows**(**C**)].
4040 If **nrows** is zero, the operation becomes a NO-OP.

4041 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**
4042 that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** - 1,
4043 then **GrB_ALL** should be specified. Regardless of execution mode and return value,
4044 this array may be manipulated by the caller after this operation returns without
4045 affecting any deferred computations for this operation. Unlike other variants, if
4046 there are duplicated values in this array the result is still defined.

4047 **ncols** (IN) The number of values in `col_indices` array. Must be in the range: $[0, \mathbf{ncols}(C)]$.
4048 If `ncols` is zero, the operation becomes a NO-OP.

4049 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
4050 should be specified. Non-default field/value pairs are listed as follows:
4051

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.

4053 Return Values

4054 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
4055 blocking mode, this indicates that the compatibility tests on di-
4056 mensions and domains for the input arguments passed successfully.
4057 Either way, output matrix C is ready to be used in the next method
4058 of the sequence.

4059 **GrB_PANIC** Unknown internal error.

4060 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
4061 GraphBLAS objects (input or output) is in an invalid state caused
4062 by a previous execution error. Call `GrB_error()` to access any error
4063 messages generated by the implementation.

4064 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

4065 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
4066 a call to `new` (or `dup` for vector parameters).

4067 **GrB_INDEX_OUT_OF_BOUNDS** A value in `row_indices` is greater than or equal to `nrows(C)`, or a
4068 value in `col_indices` is greater than or equal to `ncols(C)`. In non-
4069 blocking mode, this can be reported as an execution error.

4070 **GrB_DIMENSION_MISMATCH** Mask and C dimensions are incompatible, `nrows` is not less than
4071 `nrows(C)`, or `ncols` is not less than `ncols(C)`.

4072 **GrB_DOMAIN_MISMATCH** The domains of the matrix and scalar are incompatible with each
4073 other or the corresponding domains of the accumulation operator,
4074 or the mask's domain is not compatible with `bool`.

4075 **GrB_NULL_POINTER** Either argument `row_indices` is a NULL pointer, argument `col_indices`
4076 is a NULL pointer, or both.

4077 **Description**

4078 This variant of `GrB_assign` computes the result of assigning a constant scalar value to locations
4079 in a destination GraphBLAS matrix: $C(\text{row_indices}, \text{col_indices}) = \text{val}$; or, if an optional binary
4080 accumulation operator (\odot) is provided, $C(\text{row_indices}, \text{col_indices}) = w(\text{row_indices}, \text{col_indices}) \odot \text{val}$.
4081 More explicitly:

$$\begin{aligned} & C(\text{row_indices}[i], \text{col_indices}[j]) = \text{val}, \text{ or} \\ 4082 \quad & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4083 Logically, this operation occurs in three steps:

4084 Setup The internal vectors and mask used in the computation are formed and their domains
4085 and dimensions are tested for compatibility.

4086 Compute The indicated computations are carried out.

4087 Output The result is written into the output matrix, possibly under control of a mask.

4088 Up to two argument matrices are used in the `GrB_assign` operation:

- 4089 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
4090 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)

4091 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain
4092 compatibility as follows:

- 4093 1. The domain of `Mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
4094 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(\text{val})$.
4095 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4096 of the accumulation operator and $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4097 mulation operator.

4098 Two domains are compatible with each other if values from one domain can be cast to values in
4099 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
4100 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4101 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
4102 error listed above is returned.

4103 From the arguments, the internal matrices, index arrays, and mask used in the computation are
4104 formed (\leftarrow denotes copy):

- 4105 1. Matrix $\tilde{C} \leftarrow C$.

- 4106 2. Two-dimensional mask $\widetilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
- 4107 (a) If `Mask = GrB_NULL`, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
4108 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 4109 (b) Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) =$
4110 $\mathbf{true}\} \rangle$.
- 4111 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$.
- 4112 3. The internal row index array, $\widetilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 4113 (a) If `row_indices = GrB_ALL`, then $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
- 4114 (b) Otherwise, $\widetilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 4115 4. The internal column index array, $\widetilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 4116 (a) If `col_indices = GrB_ALL`, then $\widetilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
- 4117 (b) Otherwise, $\widetilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4118 The internal matrix and mask are checked for dimension compatibility. The following conditions
4119 must hold:

- 4120 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$.
- 4121 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$.
- 4122 3. $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\widetilde{\mathbf{C}})$.
- 4123 4. $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\widetilde{\mathbf{C}})$.

4124 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
4125 error listed above is returned.

4126 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4127 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4128 We are now ready to carry out the assign and any additional associated operations. We describe
4129 this in terms of two intermediate vectors:

- 4130 • $\widetilde{\mathbf{T}}$: The matrix holding the copies of the scalar `val` in their destination locations relative to
4131 $\widetilde{\mathbf{C}}$.
- 4132 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4133 The intermediate matrix, $\widetilde{\mathbf{T}}$, is created as follows:

4134
$$\widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(\widetilde{\mathbf{I}}[i], \widetilde{\mathbf{J}}[j], \mathbf{val}) \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle$$

4135 If either $\tilde{\mathbf{I}}$ or $\tilde{\mathbf{J}}$ is empty, this operation results in an empty matrix, $\tilde{\mathbf{T}}$. Otherwise, if any value
4136 in the $\tilde{\mathbf{I}}$ array is not in the range $[0, \mathbf{nrows}(\tilde{\mathbf{C}}))$ or any value in the $\tilde{\mathbf{J}}$ array is not in the range
4137 $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-bounds error listed above is
4138 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating
4139 `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from this point forward in the
4140 sequence.

4141 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 4142 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$4143 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4144 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l], \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4145 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
4146 of $\tilde{\mathbf{C}}$ ($\mathbf{ind}(\tilde{\mathbf{C}})$) and remove from it all the indices of $\tilde{\mathbf{C}}$ that are in the set of indices being
4147 assigned ($\{\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l], \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

4148 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4149 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4150 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l], \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4151 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \\ 4152$$

4153 where the difference operator refers to set difference. We note that, in this particular case of
4154 assigning a constant to a matrix, the sets $\{\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l], \forall k, l\}$ and $\mathbf{ind}(\tilde{\mathbf{T}})$ are identical.

- 4155 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4156 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4157 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4158 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4159 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4160 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4161 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4162 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4163$$

4164 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4165 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
4166 using what is called a *standard matrix mask and replace*. This is carried out under control of the
4167 mask which acts as a “write mask”.

- 4168 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
4169 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4170 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4171 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 4172 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 4173 mask are unchanged:

$$4174 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(-\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4175 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 4176 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 4177 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 4178 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4179 sequence.

4180 4.3.8 apply: Apply a unary function to the elements of an object

4181 Computes the transformation of the values of the elements of a vector or a matrix using a unary
 4182 function.

4183 4.3.8.1 apply: Vector variant

4184 Computes the transformation of the values of the elements of a vector using a unary function.

4185 C Syntax

```
4186     GrB_Info GrB_apply(GrB_Vector      w,
4187                       const GrB_Vector mask,
4188                       const GrB_BinaryOp accum,
4189                       const GrB_UnaryOp op,
4190                       const GrB_Vector u,
4191                       const GrB_Descriptor desc);
```

4192 Parameters

4193 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 4194 that may be accumulated with the result of the apply operation. On output, this
 4195 vector holds the results of the operation.

4196 **mask** (IN) An optional “write” mask that controls which results from this operation are
 4197 stored into the output vector **w**. The mask dimensions must match those of the
 4198 vector **w** and the domain of the **mask** vector must be of type **bool** or any of the
 4199 predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with
 4200 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

4201 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
 4202 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 4203 specified.

4204
4205
4206
4207
4208

- 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.

4209

4210 Return Values

4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224
4225
4226
4227

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.

4228 Description

4229 This variant of GrB_apply computes the result of applying a unary function to the elements of a
4230 GraphBLAS vector: $w = f(u)$; or, if an optional binary accumulation operator (\odot) is provided,
4231 $w = w \odot f(u)$.

4232 Logically, this operation occurs in three steps:

4233 **Setup** The internal vectors and mask used in the computation are formed and their domains
4234 and dimensions are tested for compatibility.

4235 **Compute** The indicated computations are carried out.

4236 **Output** The result is written into the output vector, possibly under control of a mask.

4237 Up to three argument vectors are used in this `GrB_apply` operation:

- 4238 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4239 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4240 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4241 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
4242 bility as follows:

- 4243 1. The domain of `mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.
- 4244 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the unary operator.
- 4245 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
4246 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the unary operator must be compatible with
4247 $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accumulation operator.
- 4248 4. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in}(\mathbf{op})$.

4249 Two domains are compatible with each other if values from one domain can be cast to values in
4250 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
4251 all compatible with each other. A domain from a user-defined type is only compatible with itself.
4252 If any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
4253 error listed above is returned.

4254 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
4255 denotes copy):

- 4256 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 4257 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 4258 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 4259 (b) Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 4260 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 4261 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4262 The internal vectors and masks are checked for dimension compatibility. The following conditions
4263 must hold:

4264 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$

4265 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

4266 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
4267 error listed above is returned.

4268 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4269 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4270 We are now ready to carry out the apply and any additional associated operations. We describe
4271 this in terms of two intermediate vectors:

- 4272 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the unary operator to the input vector $\tilde{\mathbf{u}}$.
- 4273 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4274 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4275 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \mathbf{L}(\tilde{\mathbf{t}}) = \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

4276 where $f = \mathbf{f}(\mathbf{op})$.

4277 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 4278 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 4279 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4280 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4281 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4282 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 4283 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 4284 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 4285 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

4286
4287
4288 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

4289 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
4290 using what is called a *standard vector mask and replace*. This is carried out under control of the
4291 mask which acts as a “write mask”.

- 4292 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
4293 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4294 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

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

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

In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of vector \mathbf{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 \mathbf{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.

4326
4327
4328

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.

4329

4330 Return Values

4331
4332
4333
4334
4335

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.

4336

GrB_PANIC Unknown internal error.

4337
4338
4339
4340

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.

4341

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

4342
4343

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).

4344
4345
4346

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.

4347
4348

GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrow \neq nrow(C), or ncol \neq ncol(C).

4349
4350
4351

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.

4352 Description

4353
4354
4355

This variant of GrB_apply computes the result of applying a unary function to the elements of a GraphBLAS matrix: $C = f(A)$; or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot f(A)$.

4356 Logically, this operation occurs in three steps:

4357 **Setup** The internal matrices and mask used in the computation are formed and their domains
4358 and dimensions are tested for compatibility.

4359 **Compute** The indicated computations are carried out.

4360 **Output** The result is written into the output matrix, possibly under control of a mask.

4361 Up to three argument matrices are used in the `GrB_apply` operation:

4362 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

4363 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)

4364 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4365 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
4366 ibility as follows:

4367 1. The domain of `Mask` (if not `GrB_NULL`) must be from one of the pre-defined types of Table 2.2.

4368 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.

4369 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4370 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
4371 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

4372 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in}(\text{op})$ of the unary operator.

4373 Two domains are compatible with each other if values from one domain can be cast to values in
4374 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
4375 all compatible with each other. A domain from a user-defined type is only compatible with itself.
4376 If any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
4377 error listed above is returned.

4378 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
4379 are formed (\leftarrow denotes copy):

4380 1. Matrix $\tilde{C} \leftarrow C$.

4381 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:

4382 (a) If `Mask = GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
4383 $j < \mathbf{ncols}(C)\} \rangle$.

4384 (b) Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) : (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.

4385 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.

4386 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

4387 The internal matrices and mask are checked for dimension compatibility. The following conditions
4388 must hold:

4389 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.

4390 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.

4391 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

4392 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

4393 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
4394 error listed above is returned.

4395 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4396 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4397 We are now ready to carry out the apply and any additional associated operations. We describe
4398 this in terms of two intermediate matrices:

- 4399 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the unary operator to the input matrix $\tilde{\mathbf{A}}$.
- 4400 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4401 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4402 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \mathbf{L}(\tilde{\mathbf{T}}) = \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

4403 where $f = \mathbf{f}(\text{op})$.

4404 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4405 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 4406 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4407 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4408 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4409 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4410 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4411 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4412 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4413 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4414 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4415 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4416 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4417 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4418 mask which acts as a “write mask”.

- 4419 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 4420 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4421 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4422 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 4423 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 4424 mask are unchanged:

$$4425 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4426 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 4427 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 4428 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 4429 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4430 sequence.

4431 4.3.9 reduce: Perform a reduction across the elements of an object

4432 Computes the reduction of the values of the elements of a vector or matrix.

4433 4.3.9.1 reduce: Standard matrix to vector variant

4434 This performs a reduction across rows of a matrix to produce a vector. If column reduction
 4435 across columns is desired, the input matrix should be transposed which can be specified using the
 4436 descriptor.

4437 C Syntax

```
4438     GrB_Info GrB_reduce(GrB_Vector      w,
4439                       const GrB_Vector mask,
4440                       const GrB_BinaryOp accum,
4441                       const GrB_Monoid  op,
4442                       const GrB_Matrix  A,
4443                       const GrB_Descriptor desc);
```

```
4444
4445     GrB_Info GrB_reduce(GrB_Vector      w,
4446                       const GrB_Vector mask,
4447                       const GrB_BinaryOp accum,
4448                       const GrB_BinaryOp op,
4449                       const GrB_Matrix  A,
4450                       const GrB_Descriptor desc);
```

4451 **Parameters**

4452 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 4453 that may be accumulated with the result of the reduction operation. On output,
 4454 this vector holds the results of the operation.

4455 **mask** (IN) An optional “write” mask that controls which results from this operation are
 4456 stored into the output vector **w**. The mask dimensions must match those of the
 4457 vector **w** and the domain of the **mask** vector must be of type **bool** or any of the
 4458 predefined “built-in” types in Table 2.2. If the default vector is desired (i.e., with
 4459 correct dimensions and filled with **true**), **GrB_NULL** should be specified.

4460 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
 4461 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 4462 specified.

4463 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.
 4464 Depending on which type is passed, the following defines the binary operator with
 4465 one domain, $F_b = \langle D, D, D, \oplus \rangle$, that is used:

4466 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.
 4467 Monoid: $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$, the identity element of the
 4468 monoid is ignored.

4469 If **op** is a **GrB_BinaryOp**, then all its domains must be the same. Furthermore, in
 4470 both cases $\odot(\text{op})$ must be commutative and associative. Otherwise, the outcome
 4471 of the operation is undefined.

4472 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

4473 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 4474 should be specified. Non-default field/value pairs are listed as follows:

4475

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.

4476

4477 **Return Values**

4478 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 4479 blocking mode, this indicates that the compatibility tests on di-
 4480 mensions and domains for the input arguments passed successfully.
 4481 Either way, output vector **w** is ready to be used in the next method
 4482 of the sequence.

4483 GrB_PANIC Unknown internal error.

4484 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
4485 GraphBLAS objects (input or output) is in an invalid state caused
4486 by a previous execution error. Call GrB_error() to access any error
4487 messages generated by the implementation.

4488 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

4489 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
4490 a call to new (or dup for vector parameters).

4491 GrB_DIMENSION_MISMATCH mask, w and/or u dimensions are incompatible.

4492 GrB_DOMAIN_MISMATCH Either the domains of the various vectors and matrices are incom-
4493 compatible with the corresponding domains of the accumulating operation,
4494 mask, and reduce function, or the domains of the GraphBLAS
4495 binary operator op are not all the same.

4496 Description

4497 This variant of GrB_reduce computes the result of performing a reduction across each of the rows
4498 of an input matrix: $w(i) = \bigoplus A(i, :)\forall i$; or, if an optional binary accumulation operator is provided,
4499 $w(i) = w(i) \odot (\bigoplus A(i, :))\forall i$, where $\bigoplus = \odot(F_b)$ and $\odot = \odot(\text{accum})$.

4500 Logically, this operation occurs in three steps:

4501 **Setup** The internal vector, matrix and mask used in the computation are formed and their
4502 domains and dimensions are tested for compatibility.

4503 **Compute** The indicated computations are carried out.

4504 **Output** The result is written into the output vector, possibly under control of a mask.

4505 Up to two vector and one matrix argument are used in this GrB_reduce operation:

- 4506 1. $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4507 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4508 3. $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4509 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested
4510 for domain compatibility as follows:

- 4511 1. The domain of mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
- 4512 2. $\mathbf{D}(w)$ must be compatible with the domain of the reduction binary operator, $\mathbf{D}(F_b)$.

- 4513 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4514 of the accumulation operator and $\mathbf{D}(F_b)$, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4515 mulation operator.
- 4516 4. $\mathbf{D}(A)$ must be compatible with the domain of the binary reduction operator, $\mathbf{D}(F_b)$.

4517 Two domains are compatible with each other if values from one domain can be cast to values in
4518 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
4519 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4520 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
4521 error listed above is returned.

4522 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
4523 denotes copy):

- 4524 1. Vector $\tilde{w} \leftarrow w$.
- 4525 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
- 4526 (a) If `mask = GrB_NULL`, then $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$.
- 4527 (b) Otherwise, $\tilde{m} = \langle \text{size}(\text{mask}), \{i : (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 4528 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{m} \leftarrow \neg\tilde{m}$.
- 4529 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

4530 The internal vectors and masks are checked for dimension compatibility. The following conditions
4531 must hold:

- 4532 1. $\text{size}(\tilde{w}) = \text{size}(\tilde{m})$
- 4533 2. $\text{size}(\tilde{w}) = \text{nrows}(\tilde{A})$.

4534 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-
4535 match error listed above is returned.

4536 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4537 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4538 We carry out the reduce and any additional associated operations. We describe this in terms of
4539 two intermediate vectors:

- 4540 • \tilde{t} : The vector holding the result from reducing along the rows of input matrix \tilde{A} .
- 4541 • \tilde{z} : The vector holding the result after application of the (optional) accumulation operator.

4542 The intermediate vector, \tilde{t} , is created as follows:

4543
$$\tilde{t} = \langle \mathbf{D}(\text{op}), \text{size}(\tilde{w}), \mathbf{L}(\tilde{t}) = \{(i, t_i) : \mathbf{ind}(A(i, :)) \neq \emptyset\} \rangle.$$

4544 The value of each of its elements is computed by

$$4545 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i,:))} \tilde{\mathbf{A}}(i, j),$$

4546 where $\bigoplus = \odot(F_b)$.

4547 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 4548 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 4549 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4550 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4551 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4552 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4553 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$4554 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$4555 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

4558 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4559 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 4560 using what is called a *standard vector mask and replace*. This is carried out under control of the
 4561 mask which acts as a “write mask”.

- 4562 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 4563 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4564 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 4565 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 4566 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 4567 mask are unchanged:

$$4568 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4569 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 4570 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 4571 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 4572 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4573 **4.3.9.2 reduce: Vector-scalar variant**

4574 Reduce all stored values into a single scalar.

4575 C Syntax

```
4576         GrB_Info GrB_reduce(<type>          *val,  
4577                               const GrB_BinaryOp  accum,  
4578                               const GrB_Monoid    op,  
4579                               const GrB_Vector    u,  
4580                               const GrB_Descriptor desc);
```

4581 Parameters

4582 **val** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
4583 a value that may be accumulated with the result of the reduction operation. On
4584 output, this scalar holds the results of the operation.

4585 **accum** (IN) An optional binary operator used for accumulating entries into existing **val**
4586 value. If assignment rather than accumulation is desired, **GrB_NULL** should be
4587 specified.

4588 **op** (IN) The monoid used in the element-wise reduction operation, $M = \langle D, \oplus, 0 \rangle$.
4589 The binary operator, \oplus , must be commutative and associative; otherwise, the
4590 outcome of the operation is undefined.

4591 **u** (IN) The GraphBLAS vector on which reduction will be performed.

4592 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4593 should be specified. Non-default field/value pairs are listed as follows:

4594

4595	Param	Field	Value	Description
------	-------	-------	-------	-------------

4596 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
4597 tions. There are currently no non-default field/value pairs that can be set for this
4598 operation.

4599 Return Values

4600 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
4601 cessfully, and the output scalar **val** is ready to be used in the next
4602 method of the sequence.

4603 **GrB_PANIC** Unknown internal error.

4604 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
4605 GraphBLAS objects (input or output) is in an invalid state caused
4606 by a previous execution error. Call **GrB_error()** to access any error
4607 messages generated by the implementation.

4608 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

4609 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
4610 a call to new (or Vector_dup for vector parameters).

4611 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
4612 the corresponding domains of the accumulation operator, or reduce
4613 operator.

4614 GrB_NULL_POINTER val pointer is NULL.

4615 Description

4616 This variant of GrB_reduce computes the result of performing a reduction across each of the elements
4617 of an input vector: $\text{val} = \bigoplus u(:)$; or, if an optional binary accumulation operator is provided,
4618 $\text{val} = \text{val} \odot (\bigoplus u(:))$, where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

4619 Logically, this operation occurs in three steps:

4620 **Setup** The internal vector used in the computation is formed and its domain is tested for
4621 compatibility.

4622 **Compute** The indicated computations are carried out.

4623 **Output** The result is written into the output scalar.

4624 One vector argument is used in this GrB_reduce operation:

4625 1. $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4626 The output scalar, argument vector, reduction operator and accumulation operator (if provided)
4627 are tested for domain compatibility as follows:

4628 1. If accum is GrB_NULL, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}(\text{op})$ of the reduction binary
4629 operator.

4630 2. If accum is not GrB_NULL, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4631 of the accumulation operator and $\mathbf{D}(\text{op})$ of the reduction binary operator must be compatible
4632 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

4633 3. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}(\text{op})$ of the binary reduction operator.

4634 Two domains are compatible with each other if values from one domain can be cast to values in
4635 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
4636 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4637 any compatibility rule above is violated, execution of GrB_reduce ends and the domain mismatch
4638 error listed above is returned.

4639 From the argument vector, the internal vector used in the computation is formed (\leftarrow denotes copy):

4640 1. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4641 We are now ready to carry out the reduce and any additional associated operations. First, an
4642 intermediate scalar result t is computed using the recurrence:

$$4643 \quad t \leftarrow \mathbf{0}(\text{op}),$$

$$4644 \quad t \leftarrow t \oplus \mathbf{u}(i), \forall i \in \mathbf{ind}(\mathbf{u}).$$

4646 Where $\oplus = \odot(\text{op})$, and $\mathbf{0}(\text{op})$ is the identity of the monoid.

4647 The final reduction value val is computed as follows:

- 4648 • If $\text{accum} = \text{GrB_NULL}$, then $\text{val} \leftarrow t$.
- 4649 • If accum is a binary operator, then $\text{val} \leftarrow \text{val} \odot t$, where $\odot = \odot(\text{accum})$.

4650 In both `GrB_BLOCKING` and `GrB_NONBLOCKING` modes, the method exits with return value
4651 `GrB_SUCCESS` and the new contents of val is as defined above and fully computed.

4652 4.3.9.3 reduce: Matrix-scalar variant

4653 Reduce all stored values into a single scalar.

4654 C Syntax

```
4655     GrB_Info GrB_reduce(<type>          *val,  
4656                          const GrB_BinaryOp accum,  
4657                          const GrB_Monoid   op,  
4658                          const GrB_Matrix   A,  
4659                          const GrB_Descriptor desc);
```

4660 Parameters

4661 **val** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
4662 a value that may be accumulated with the result of the reduction operation. On
4663 output, this scalar holds the results of the operation.

4664 **accum** (IN) An optional binary operator used for accumulating entries into existing **val**
4665 value. If assignment rather than accumulation is desired, `GrB_NULL` should be
4666 specified.

4667 **op** (IN) The monoid used in the element-wise reduction operation, $M = \langle D, \oplus, 0 \rangle$.
4668 The binary operator, \oplus , must be commutative and associative; otherwise, the
4669 outcome of the operation is undefined.

4670 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

4671 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
4672 should be specified. Non-default field/value pairs are listed as follows:

4673

4674

Param	Field	Value	Description
-------	-------	-------	-------------

4675 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
4676 tions. There are currently no non-default field/value pairs that can be set for this
4677 operation.

4678 Return Values

4679 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
4680 cessfully, and the output scalar *val* is ready to be used in the next
4681 method of the sequence.

4682 GrB_PANIC Unknown internal error.

4683 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
4684 GraphBLAS objects (input or output) is in an invalid state caused
4685 by a previous execution error. Call GrB_error() to access any error
4686 messages generated by the implementation.

4687 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

4688 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
4689 a call to new (or Matrix_dup for matrix parameters).

4690 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
4691 the corresponding domains of the accumulation operator, or reduce
4692 operator.

4693 GrB_NULL_POINTER *val* pointer is NULL.

4694 Description

4695 This variant of GrB_reduce computes the result of performing a reduction across each of the elements
4696 of an input matrix: $\text{val} = \bigoplus A(:, :)$; or, if an optional binary accumulation operator is provided,
4697 $\text{val} = \text{val} \odot (\bigoplus A(:, :))$, where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

4698 Logically, this operation occurs in three steps:

4699 **Setup** The internal matrix used in the computation is formed and its domain is tested for
4700 compatibility.

4701 **Compute** The indicated computations are carried out.

4702 **Output** The result is written into the output scalar.

4703 One matrix argument is used in this GrB_reduce operation:

4704 1. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{size}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{i,j})\} \rangle$

4705 The output scalar, argument matrix, reduction operator and accumulation operator (if provided)
4706 are tested for domain compatibility as follows:

4707 1. If `accum` is `GrB_NULL`, then $\mathbf{D}(\mathbf{val})$ must be compatible with $\mathbf{D}(\mathbf{op})$ of the reduction binary
4708 operator.

4709 2. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{val})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
4710 of the accumulation operator and $\mathbf{D}(\mathbf{op})$ of the reduction binary operator must be compatible
4711 with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accumulation operator.

4712 3. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}(\mathbf{op})$ of the binary reduction operator.

4713 Two domains are compatible with each other if values from one domain can be cast to values in
4714 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
4715 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4716 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
4717 error listed above is returned.

4718 From the argument matrix, the internal matrix used in the computation is formed (\leftarrow denotes
4719 copy):

4720 1. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{A}$.

4721 We are now ready to carry out the reduce and any additional associated operations. First, an
4722 intermediate scalar result t is computed using the recurrence:

$$\begin{aligned} 4723 \quad t &\leftarrow \mathbf{0}(\mathbf{op}), \\ 4724 \quad t &\leftarrow t \oplus \mathbf{A}(i, j), \forall (i, j) \in \mathbf{ind}(\mathbf{A}). \end{aligned}$$

4726 Where $\oplus = \odot(\mathbf{op})$, and $\mathbf{0}(\mathbf{op})$ is the identity of the monoid.

4727 The final reduction value `val` is computed as follows:

- 4728 • If `accum` = `GrB_NULL`, then $\mathbf{val} \leftarrow t$.
- 4729 • If `accum` is a binary operator, then $\mathbf{val} \leftarrow \mathbf{val} \odot t$, where $\odot = \odot(\mathbf{accum})$.

4730 In both `GrB_BLOCKING` and `GrB_NONBLOCKING` modes, the method exits with return value
4731 `GrB_SUCCESS` and the new contents of `val` is as defined above and fully computed.

4732 4.3.10 transpose: Transpose rows and columns of a matrix

4733 This version computes a new matrix that is the transpose of the source matrix.

4734 **C Syntax**

```

4735         GrB_Info GrB_transpose(GrB_Matrix      C,
4736                               const GrB_Matrix  Mask,
4737                               const GrB_BinaryOp accum,
4738                               const GrB_Matrix  A,
4739                               const GrB_Descriptor desc);

```

4740 **Parameters**

4741 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
4742 that may be accumulated with the result of the transpose operation. On output,
4743 the matrix holds the results of the operation.

4744 **Mask** (IN) An optional “write” mask that controls which results from this operation are
4745 stored into the output matrix C. The mask dimensions must match those of the
4746 matrix C and the domain of the Mask matrix must be of type bool or any of the
4747 predefined “built-in” types in Table 2.2. If the default matrix is desired (i.e., with
4748 correct dimensions and filled with true), GrB_NULL should be specified.

4749 **accum** (IN) An optional binary operator used for accumulating entries into existing C
4750 entries. If assignment rather than accumulation is desired, GrB_NULL should be
4751 specified.

4752 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

4753 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
4754 should be specified. Non-default field/value pairs are listed as follows:
4755

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.

4757 **Return Values**

4758 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
4759 blocking mode, this indicates that the compatibility tests on di-
4760 mensions and domains for the input arguments passed successfully.
4761 Either way, output matrix C is ready to be used in the next method
4762 of the sequence.

4763 **GrB_PANIC** Unknown internal error.

4764 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
4765 GraphBLAS objects (input or output) is in an invalid state caused
4766 by a previous execution error. Call GrB_error() to access any error
4767 messages generated by the implementation.

4768 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

4769 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
4770 a call to new (or Matrix_dup for matrix parameters).

4771 GrB_DIMENSION_MISMATCH mask, C and/or A dimensions are incompatible.

4772 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
4773 corresponding domains of the accumulation operator, or the mask's
4774 domain is not compatible with bool.

4775 Description

4776 GrB_transpose computes the result of performing a transpose of the input matrix: $C = A^T$; or, if an
4777 optional binary accumulation operator (\odot) is provided, $C = C \odot A^T$. We note that the input matrix
4778 A can itself be optionally transposed before the operation, which would cause either an assignment
4779 from A to C or an accumulation of A into C.

4780 Logically, this operation occurs in three steps:

4781 **Setup** The internal matrix and mask used in the computation are formed and their domains
4782 and dimensions are tested for compatibility.

4783 **Compute** The indicated computations are carried out.

4784 **Output** The result is written into the output matrix, possibly under control of a mask.

4785 Up to three matrix arguments are used in this GrB_transpose operation:

- 4786 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4787 2. $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 4788 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4789 The argument matrices and accumulation operator (if provided) are tested for domain compatibility
4790 as follows:

- 4791 1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.2.
- 4792 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$ of the input matrix.
- 4793 3. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4794 of the accumulation operator and $\mathbf{D}(A)$ of the input matrix must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
4795 of the accumulation operator.

4796 Two domains are compatible with each other if values from one domain can be cast to values in
 4797 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 4798 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4799 any compatibility rule above is violated, execution of `GrB_transpose` ends and the domain mismatch
 4800 error listed above is returned.

4801 From the argument matrices, the internal matrices and mask used in the computation are formed
 4802 (\leftarrow denotes copy):

- 4803 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4804 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 4805 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 4806 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 4807 (b) Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) =$
 4808 $\mathbf{true}\} \rangle$.
 - 4809 (c) If `desc[GrB_MASK].GrB_SCMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4810 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

4811 The internal matrices and masks are checked for dimension compatibility. The following conditions
 4812 must hold:

- 4813 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 4814 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 4815 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.
- 4816 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

4817 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension
 4818 mismatch error listed above is returned.

4819 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4820 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4821 We are now ready to carry out the matrix transposition and any additional associated operations.
 4822 We describe this in terms of two intermediate matrices:

- 4823 • $\tilde{\mathbf{T}}$: The matrix holding the transpose of $\tilde{\mathbf{A}}$.
- 4824 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4825 The intermediate matrix

$$4826 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\tilde{\mathbf{A}}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{L}(\tilde{\mathbf{T}}) = \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$$

4827 is created.

4828 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4829 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 4830 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4831 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4832 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4833 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4834 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4835 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4837 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4839 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4840 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 4841 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 4842 mask which acts as a “write mask”.

- 4843 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 4844 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4845 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4846 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 4847 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 4848 mask are unchanged:

$$4849 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4850 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4851 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4852 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 4853 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 4854 sequence.

4855 4.4 Sequence Termination

4856 4.4.1 wait: Waits until pending operations complete

4857 When running in non-blocking mode, this function guarantees that all pending GraphBLAS opera-
 4858 tions are fully executed. Note that this can be called in blocking mode without an error, but there
 4859 should be no pending GraphBLAS operations to complete.

4860 C Syntax

```
4861         GrB_Info GrB_wait();
```

4862 Parameters

4863 Return values

4864 `GrB_SUCCESS` operation completed successfully.

4865 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happend during completion
4866 of pending operations.

4867 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion of
4868 pending operations.

4869 `GrB_PANIC` unknown internal error.

4870 Description

4871 Upon successful return, all previously called GraphBLAS methods have fully completed their exe-
4872 cution, and any (transparent or opaque) data structures produced or manipulated by those methods
4873 can be safely touched. If an error occured in any pending GraphBLAS operations, `GrB_error()` can
4874 be used to retrieve implementation defined error information about the problem encountered.

4875 4.4.2 error: Get an error message regarding internal errors

```
4876         const char *GrB_error();
```

4877 Parameters

4878 Return value

- 4879 • A pointer to a null-terminated string (owned by the library).

4880 Description

4881 After a call to any GraphBLAS method, the program can retrieve additional error information
4882 (beyond the error code returned by the method) though a call to the function `GrB_error()`. The
4883 function returns a pointer to a null terminated string and the contents of that string are implemen-
4884 tation dependent. In particular, a null string (not a `NULL` pointer) is always a valid error string.
4885 The pointer is valid until the next call to any GraphBLAS method by the same thread. `GrB_error()`
4886 is a thread-safe function, in the sense that multiple threads can call it simultaneously and each will
4887 get its own error string back, referring to the last GraphBLAS method it called.

4888 **Chapter 5**

4889 **Nonpolymorphic Interface**

4890 Each polymorphic GraphBLAS method (those with multiple parameter signatures under the same
 4891 name) has a corresponding set of long-name forms that are specific to each parameter signature.
 4892 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*,..., <i>other</i>)	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
<code>GrB_extract(GrB_Vector,...,GrB_Vector,...)</code>	<code>GrB_Vector_extract(GrB_Vector,...,GrB_Vector,...)</code>
<code>GrB_extract(GrB_Matrix,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_extract(GrB_Matrix,...,GrB_Matrix,...)</code>
<code>GrB_extract(GrB_Vector,...,GrB_Matrix,...)</code>	<code>GrB_Col_extract(GrB_Vector,...,GrB_Matrix,...)</code>
<code>GrB_assign(GrB_Vector,...,GrB_Vector,...)</code>	<code>GrB_Vector_assign(GrB_Vector,...,GrB_Vector,...)</code>
<code>GrB_assign(GrB_Matrix,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_assign(GrB_Matrix,...,GrB_Matrix,...)</code>
<code>GrB_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)</code>	<code>GrB_Col_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)</code>
<code>GrB_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)</code>	<code>GrB_Row_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)</code>
<code>GrB_assign(GrB_Vector,...,bool,...)</code>	<code>GrB_Vector_assign_BOOL(GrB_Vector,...,bool,...)</code>
<code>GrB_assign(GrB_Vector,...,int8_t,...)</code>	<code>GrB_Vector_assign_INT8(GrB_Vector,...,int8_t,...)</code>
<code>GrB_assign(GrB_Vector,...,uint8_t,...)</code>	<code>GrB_Vector_assign_UINT8(GrB_Vector,...,uint8_t,...)</code>
<code>GrB_assign(GrB_Vector,...,int16_t,...)</code>	<code>GrB_Vector_assign_INT16(GrB_Vector,...,int16_t,...)</code>
<code>GrB_assign(GrB_Vector,...,uint16_t,...)</code>	<code>GrB_Vector_assign_UINT16(GrB_Vector,...,uint16_t,...)</code>
<code>GrB_assign(GrB_Vector,...,int32_t,...)</code>	<code>GrB_Vector_assign_INT32(GrB_Vector,...,int32_t,...)</code>
<code>GrB_assign(GrB_Vector,...,uint32_t,...)</code>	<code>GrB_Vector_assign_UINT32(GrB_Vector,...,uint32_t,...)</code>
<code>GrB_assign(GrB_Vector,...,int64_t,...)</code>	<code>GrB_Vector_assign_INT64(GrB_Vector,...,int64_t,...)</code>
<code>GrB_assign(GrB_Vector,...,uint64_t,...)</code>	<code>GrB_Vector_assign_UINT64(GrB_Vector,...,uint64_t,...)</code>
<code>GrB_assign(GrB_Vector,...,float,...)</code>	<code>GrB_Vector_assign_FP32(GrB_Vector,...,float,...)</code>
<code>GrB_assign(GrB_Vector,...,double,...)</code>	<code>GrB_Vector_assign_FP64(GrB_Vector,...,double,...)</code>
<code>GrB_assign(GrB_Vector,...,other,...)</code>	<code>GrB_Vector_assign_UDT(GrB_Vector,...,const void*,...)</code>
<code>GrB_assign(GrB_Matrix,...,bool,...)</code>	<code>GrB_Matrix_assign_BOOL(GrB_Matrix...,bool,...)</code>
<code>GrB_assign(GrB_Matrix,...,int8_t,...)</code>	<code>GrB_Matrix_assign_INT8(GrB_Matrix...,int8_t,...)</code>
<code>GrB_assign(GrB_Matrix,...,uint8_t,...)</code>	<code>GrB_Matrix_assign_UINT8(GrB_Matrix...,uint8_t,...)</code>
<code>GrB_assign(GrB_Matrix,...,int16_t,...)</code>	<code>GrB_Matrix_assign_INT16(GrB_Matrix...,int16_t,...)</code>
<code>GrB_assign(GrB_Matrix,...,uint16_t,...)</code>	<code>GrB_Matrix_assign_UINT16(GrB_Matrix...,uint16_t,...)</code>
<code>GrB_assign(GrB_Matrix,...,int32_t,...)</code>	<code>GrB_Matrix_assign_INT32(GrB_Matrix...,int32_t,...)</code>
<code>GrB_assign(GrB_Matrix,...,uint32_t,...)</code>	<code>GrB_Matrix_assign_UINT32(GrB_Matrix...,uint32_t,...)</code>
<code>GrB_assign(GrB_Matrix,...,int64_t,...)</code>	<code>GrB_Matrix_assign_INT64(GrB_Matrix...,int64_t,...)</code>
<code>GrB_assign(GrB_Matrix,...,uint64_t,...)</code>	<code>GrB_Matrix_assign_UINT64(GrB_Matrix...,uint64_t,...)</code>
<code>GrB_assign(GrB_Matrix,...,float,...)</code>	<code>GrB_Matrix_assign_FP32(GrB_Matrix...,float,...)</code>
<code>GrB_assign(GrB_Matrix,...,double,...)</code>	<code>GrB_Matrix_assign_FP64(GrB_Matrix...,double,...)</code>
<code>GrB_assign(GrB_Matrix,...,other,...)</code>	<code>GrB_Matrix_assign_UDT(GrB_Matrix...,const void*,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_Vector,...)</code>	<code>GrB_Vector_apply(GrB_Vector,...,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply(GrB_Matrix,...,GrB_Matrix,...)</code>

Table 5.6: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
<code>GrB_reduce(GrB_Vector,...,GrB_Monoid,...)</code>	<code>GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)</code>
<code>GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)</code>	<code>GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)</code>
<code>GrB_reduce(bool*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)</code>
<code>GrB_reduce(int8_t*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)</code>
<code>GrB_reduce(uint8_t*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)</code>
<code>GrB_reduce(int16_t*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)</code>
<code>GrB_reduce(uint16_t*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)</code>
<code>GrB_reduce(int32_t*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)</code>
<code>GrB_reduce(uint32_t*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)</code>
<code>GrB_reduce(int64_t*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)</code>
<code>GrB_reduce(uint64_t*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)</code>
<code>GrB_reduce(float*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)</code>
<code>GrB_reduce(double*,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)</code>
<code>GrB_reduce(<i>other</i>,...,GrB_Vector,...)</code>	<code>GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)</code>
<code>GrB_reduce(bool*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(int8_t*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(uint8_t*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(int16_t*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(uint16_t*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(int32_t*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(uint32_t*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(int64_t*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(uint64_t*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(float*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(double*,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)</code>
<code>GrB_reduce(<i>other</i>,...,GrB_Matrix,...)</code>	<code>GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)</code>

4893 Appendix A

4894 Revision History

4895 Changes in 1.2.0:

- 4896 • Removed "provisional" clause.

4897 Changes in 1.1.0:

- 4898 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
4899 `assign` operations.
- 4900 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
- 4901 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
- 4902 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
4903 from the domains of the binary operator provided.
- 4904 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out argu-
4905 ment, `n`, which indicates the size of the output arrays provided (in terms of number of ele-
4906 ments, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE` which
4907 is returned when the capacities of the output arrays are insufficient to hold all of the tuples.
- 4908 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic interface.
- 4909 • Added replace flag (`z`) notation to Table 4.1.
- 4910 • Updated the "Mathematical Description" of the assign operation in Table 4.1.
- 4911 • Added triangle counting example.
- 4912 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
4913 sections of GraphBLAS operations when the respective text was the "standard" text (i.e.,
4914 identical in a majority of the operations).
- 4915 • Fixed typographical errors.

4916 Changes in 1.0.2:

- 4917 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
4918 matrices and avoid casting issues in certain implementations.
- 4919 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
4920 erased outside the assigned area.
- 4921 • Changes non-polymorphic interface:
 - 4922 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.
 - 4923 – Renamed `GrB_Vector_reduce_BinaryOp` to `GrB_Matrix_reduce_BinaryOp`.
 - 4924 – Renamed `GrB_Vector_reduce_Monoid` to `GrB_Matrix_reduce_Monoid`.
- 4925 • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 4926 • Fixed numerous typographical errors.

4927 **Appendix B**

4928 **Examples**

B.1 Example: breadth-first search (BFS) in GraphBLAS

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9  * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] = 1$ ).
10 * If  $i$  is not reachable from  $s$ , then  $v[i] = 0$ . (Vector  $v$  should be empty on input.)
11 */
12 GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
13 {
14     GrB_Index n;
15     GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
16
17     GrB_Vector_new(v,GrB_INT32,n);    // Vector<int32_t>  $v(n)$ 
18
19     GrB_Vector q;                     // vertices visited in each level
20     GrB_Vector_new(&q,GrB_BOOL,n);    // Vector<bool>  $q(n)$ 
21     GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
22
23     GrB_Monoid Lor;                   // Logical-or monoid
24     GrB_Monoid_new(&Lor,GrB_LOR,(bool>false));
25
26     GrB_Semiring Boolean;             // Boolean semiring
27     GrB_Semiring_new(&Boolean,Lor,GrB_LAND);
28
29     GrB_Descriptor desc;              // Descriptor for vxm
30     GrB_Descriptor_new(&desc);
31     GrB_Descriptor_set(desc,GrB_MASK,GrB_SCMP); // invert the mask
32     GrB_Descriptor_set(desc,GrB_OUTP,GrB_REPLACE); // clear the output before assignment
33
34     /*
35      * BFS traversal and label the vertices.
36      */
37     int32_t d = 0;                    //  $d =$  level in BFS traversal
38     bool succ = false;                //  $\text{succ} = \text{true}$  when some successor found
39     do {
40         ++d;                           // next level (start with 1)
41         GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL); //  $v[q] = d$ 
42         GrB_vxm(q,*v,GrB_NULL,Boolean,q,A,desc); //  $q[!v] = q \ || \ \&\& \ A$ ; finds all the
43             // unvisited successors from current  $q$ 
44         GrB_reduce(&succ,GrB_NULL,Lor,q,GrB_NULL); //  $\text{succ} = \ || \ (q)$ 
45     } while (succ);                    // if there is no successor in  $q$ , we are done.
46
47     GrB_free(&q);                       //  $q$  vector no longer needed
48     GrB_free(&Lor);                     // Logical or monoid no longer needed
49     GrB_free(&Boolean);                 // Boolean semiring no longer needed
50     GrB_free(&desc);                    // descriptor no longer needed
51
52     return GrB_SUCCESS;
53 }

```

B.2 Example: BFS in GraphBLAS using apply

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 int32_t level = 0; // level = depth in BFS traversal, roots=1, unvisited=0
8 void return_level(void *out, const void *in) {
9     bool element = *(bool*)in;
10    *(int32_t*)out = level;
11 }
12
13 /*
14  * Given a boolean n x n adjacency matrix A and a source vertex s, performs a BFS traversal
15  * of the graph and sets v[i] to the level in which vertex i is visited (v[s] == 1).
16  * If i is not reachable from s, then v[i] = 0. (Vector v should be empty on input.)
17  */
18 GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
19 {
20     GrB_Index n;
21     GrB_Matrix_nrows(&n,A); // n = # of rows of A
22
23     GrB_Vector_new(v,GrB_INT32,n); // Vector<int32_t> v(n) = 0
24
25     GrB_Vector q; // vertices visited in each level
26     GrB_Vector_new(&q,GrB_BOOL,n); // Vector<bool> q(n) = false
27     GrB_Vector_setElement(q,(bool)true,s); // q[s] = true, false everywhere else
28
29     GrB_Monoid Lor; // Logical-or monoid
30     GrB_Monoid_new(&Lor,GrB_LOR,false);
31
32     GrB_Semiring Boolean; // Boolean semiring
33     GrB_Semiring_new(&Boolean,Lor,GrB_LAND);
34
35     GrB_Descriptor desc; // Descriptor for vxm
36     GrB_Descriptor_new(&desc);
37     GrB_Descriptor_set(desc,GrB_MASK,GrB_SCMP); // invert the mask
38     GrB_Descriptor_set(desc,GrB_OUTP,GrB_REPLACE); // clear the output before assignment
39
40     GrB_UnaryOp apply_level;
41     GrB_UnaryOp_new(&apply_level,return_level,GrB_INT32,GrB_BOOL);
42
43     /*
44     * BFS traversal and label the vertices.
45     */
46     level = 0;
47     GrB_Index nvals;
48     do {
49         ++level; // next level (start with 1)
50         GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,apply_level,q,GrB_NULL); // v[q] = level
51         GrB_vxm(q,*v,GrB_NULL,Boolean,q,A,desc); // q[!v] = q ||.A; finds all the
52         // unvisited successors from current q
53         GrB_Vector_nvals(&nvals,q);
54     } while (nvals); // if there is no successor in q, we are done.
55
56     GrB_free(&q); // q vector no longer needed
57     GrB_free(&Lor); // Logical or monoid no longer needed
58     GrB_free(&Boolean); // Boolean semiring no longer needed
59     GrB_free(&desc); // descriptor no longer needed
60
61     return GrB_SUCCESS;
62 }

```

B.3 Example: betweenness centrality (BC) in GraphBLAS

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ ,
9  * compute the BC-metric vector  $\delta$ , which should be empty on input.
10 */
11 GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n,A);           //  $n = \#$  of vertices in graph
15
16     GrB_Vector_new(delta, GrB_FP32, n); // Vector<float>  $\delta(n)$ 
17
18     GrB_Matrix sigma;                // Matrix<int32_t>  $\sigma(n,n)$ 
19     GrB_Matrix_new(&sigma, GrB_INT32, n, n); //  $\sigma[d,k] = \#$  shortest paths to node  $k$  at level  $d$ 
20
21     GrB_Vector q;
22     GrB_Vector_new(&q, GrB_INT32, n); // Vector<int32_t>  $q(n)$  of path counts
23     GrB_Vector_setElement(q, 1, s); //  $q[s] = 1$ 
24
25     GrB_Vector p;                    // Vector<int32_t>  $p(n)$  shortest path counts so far
26     GrB_Vector_dup(&p, q);           //  $p = q$ 
27
28     GrB_Monoid Int32Add;              // Monoid <int32_t,+,0>
29     GrB_Monoid_new(&Int32Add, GrB_PLUS_INT32, 0);
30     GrB_Semiring Int32AddMul;        // Semiring <int32_t, int32_t, int32_t, +, *, 0, 1>
31     GrB_Semiring_new(&Int32AddMul, Int32Add, GrB_TIMES_INT32);
32
33     GrB_Descriptor desc;              // Descriptor for vxm
34     GrB_Descriptor_new(&desc);
35     GrB_Descriptor_set(desc, GrB_MASK, GrB_SCMP); // structural complement of the mask
36     GrB_Descriptor_set(desc, GrB_OUTP, GrB_REPLACE); // clear the output before assignment
37
38     GrB_Descriptor tr1;               // Transpose 1st input argument
39     GrB_Descriptor_new(&tr1);
40     GrB_Descriptor_set(tr1, GrB_INP0, GrB_TRAN); // structural complement of the mask
41
42     /*
43     * BFS phase
44     */
45     int32_t d = 0;                    // BFS level number
46     int32_t sum = 0;                  // sum == 0 when BFS phase is complete
47     do {
48         GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\sigma[d, :] = q$ 
49         GrB_vxm(q, p, GrB_NULL, Int32AddMul, q, A, desc); //  $q = \#$  paths to nodes reachable
50                                                         // from current level
51         GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, Int32AddMul, p, q, GrB_NULL); // accumulate path counts on this level
52         GrB_reduce(&sum, GrB_NULL, Int32Add, q, GrB_NULL); // sum path counts at this level
53         ++d;
54     } while (sum);
55
56     /*
57     * BC computation phase
58     * ( $t_1, t_2, t_3, t_4$ ) are temporary vectors
59     */
60     GrB_Monoid FP32Add;               // Monoid <float, float, float, +, 0.0>
61     GrB_Monoid_new(&FP32Add, GrB_PLUS_FP32, 0.0f);
62

```

```

63 GrB_Monoid FP32Mul; // Monoid <float , float , float , *, 1.0>
64 GrB_Monoid_new(&FP32Mul, GrB_TIMES_FP32, 1.0 f);
65
66 GrB_Semiring FP32AddMul; // Semiring <float , float , float , +, *, 0.0, 1.0>
67 GrB_Semiring_new(&FP32AddMul, FP32Add, GrB_TIMES_FP32);
68
69 GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
70 GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
71 GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
72 GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
73 for(int i=d-1; i>0; i--)
74 {
75     GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0 f, GrB_ALL, n, GrB_NULL); // t1 = 1+delta
76     GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, FP32Add, t1, *delta, GrB_NULL);
77     GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, tr1); // t2 = sigma[i, :]
78     GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); // t2 = (1+delta)/sigma[i, :]
79     GrB_mvx(t3, GrB_NULL, GrB_NULL, FP32AddMul, A, t2, GrB_NULL); // add contributions made by
80 // successors of a node
81     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, tr1); // t4 = sigma[i-1, :]
82     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, FP32Mul, t4, t3, GrB_NULL); // t4 = sigma[i-1, :]*t3
83     GrB_eWiseAdd(*delta, GrB_NULL, GrB_NULL, FP32Add, *delta, t4, GrB_NULL); // accumulate into delta
84 }
85
86 GrB_free(&sigma);
87 GrB_free(&q); GrB_free(&p);
88 GrB_free(&Int32AddMul); GrB_free(&Int32Add); GrB_free(&FP32AddMul);
89 GrB_free(&FP32Add); GrB_free(&FP32Mul);
90 GrB_free(&desc);
91 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
92
93 return GrB_SUCCESS;
94 }

```

B.4 Example: batched BC in GraphBLAS

```

1 #include <stdlib.h>
2 #include "GraphBLAS.h" // in addition to other required C headers
3
4 // Compute partial BC metric for a subset of source vertices, s, in graph A
5 GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
6 {
7     GrB_Index n;
8     GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
9     GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
10
11     GrB_Monoid Int32Add; // Monoid <int32_t,+,0>
12     GrB_Monoid_new(&Int32Add, GrB_PLUS_INT32, 0);
13     GrB_Semiring Int32AddMul; // Semiring <int32_t, int32_t, int32_t, +, *, 0>
14     GrB_Semiring_new(&Int32AddMul, Int32Add, GrB_TIMES_INT32);
15
16     // Descriptor for BFS phase mxm
17     GrB_Descriptor desc_tsr;
18     GrB_Descriptor_new(&desc_tsr);
19     GrB_Descriptor_set(desc_tsr, GrB_INP0, GrB_TRAN); // transpose the adjacency matrix
20     GrB_Descriptor_set(desc_tsr, GrB_MASK, GrB_SCOMP); // complement the mask
21     GrB_Descriptor_set(desc_tsr, GrB_OUTP, GrB_REPLACE); // clear output before result is stored
22
23     // index and value arrays needed to build numsp
24     GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
25     int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
26     for(int i=0; i<nsver; ++i) {
27         i_nsver[i] = i;
28         ones[i] = 1;
29     }
30
31     // numsp: structure holds the number of shortest paths for each node and starting vertex
32     // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
33     GrB_Matrix numsp;
34     GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
35     GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
36     free(i_nsver); free(ones); //
37
38     // frontier: Holds the current frontier where values are path counts.
39     // Initialized to out vertices of each source node in s.
40     GrB_Matrix frontier;
41     GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
42     GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, desc_tsr); //
43
44     // sigma: stores frontier information for each level of BFS phase. The memory
45     // for an entry in sigmas is only allocated within the do-while loop if needed
46     GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n); // n is an upper bound on diameter
47
48     int32_t d = 0; // BFS level number
49     GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
50
51     // ----- The BFS phase (forward sweep) -----
52     do {
53         // sigmas[d](:,s) = d^th level frontier from source vertex s
54         GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
55
56         GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
57                 GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,s) = (Boolean) frontier
58         GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL,
59                    Int32Add, numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
60         GrB_mxm(frontier, numsp, GrB_NULL,
61                Int32AddMul, A, frontier, desc_tsr); // f<!numsp> = A' +.* f (update frontier)
62         GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level

```

```

63     d++;
64 } while ( nvals );
65
66 GrB_Monoid FP32Add; // Monoid <float,+,0.0>
67 GrB_Monoid_new(&FP32Add, GrB_PLUS_FP32, 0.0 f);
68 GrB_Monoid FP32Mul; // Monoid <float,*,1.0>
69 GrB_Monoid_new(&FP32Mul, GrB_TIMES_FP32, 1.0 f);
70 GrB_Semiring FP32AddMul; // Semiring <float, float, float, +, *, 0.0>
71 GrB_Semiring_new(&FP32AddMul, FP32Add, GrB_TIMES_FP32);
72
73 // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
74 GrB_Matrix nspinv;
75 GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
76 GrB_apply(nspinv, GrB_NULL, GrB_NULL,
77         GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
78
79 // bcu: BC updates for each vertex for each starting vertex in s
80 GrB_Matrix bcu;
81 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
82 GrB_assign(bcu, GrB_NULL, GrB_NULL,
83         1.0 f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
84
85 // Descriptor used in the tally phase
86 GrB_Descriptor desc_r;
87 GrB_Descriptor_new(&desc_r);
88 GrB_Descriptor_set(desc_r, GrB_OUTP, GrB_REPLACE); // clear output before result is stored
89
90 GrB_Matrix w; // temporary workspace matrix
91 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
92
93 // ----- Tally phase (backward sweep) -----
94 for (int i=d-1; i>0; i--) {
95     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
96         FP32Mul, bcu, nspinv, desc_r); // w<sigmas[i]>=(1 ./ nsp).*bcu
97
98     // add contributions by successors and mask with that BFS level's frontier
99     GrB_mxm(w, sigmas[i-1], GrB_NULL,
100         FP32AddMul, A, w, desc_r); // w<sigmas[i-1]> = (A +.* w)
101     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32,
102         FP32Mul, w, numsp, GrB_NULL); // bcu += w .* numsp
103 }
104
105 // subtract "nsver" from every entry in delta (account for 1 extra value per bcu element)
106 GrB_assign(*delta, GrB_NULL, GrB_NULL,
107     -(float)nsver, GrB_ALL, n, GrB_NULL); // fill with -nsver
108 GrB_reduce(*delta, GrB_NULL, GrB_PLUS_FP32,
109     GrB_PLUS_FP32, bcu, GrB_NULL); // add all updates to -nsver
110
111 // Release resources
112 for (int i=0; i<d; i++) {
113     GrB_free(&(sigmas[i]));
114 }
115 free(sigmas);
116
117 GrB_free(&frontier); GrB_free(&numsp);
118 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
119 GrB_free(&desc_tsr); GrB_free(&desc_r);
120 GrB_free(&Int32AddMul); GrB_free(&Int32Add);
121 GrB_free(&FP32AddMul); GrB_free(&FP32Add); GrB_free(&FP32Mul);
122
123 return GrB_SUCCESS;
124 }

```

B.5 Example: maximal independent set (MIS) in GraphBLAS

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 // Assign a random number to each element scaled by the inverse of the node's degree.
8 // This will increase the probability that low degree nodes are selected and larger
9 // sets are selected.
10 void setRandom(void *out, const void *in)
11 {
12     uint32_t degree = *(uint32_t*)in;
13     *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
14 }
15
16 /*
17 * A variant of Luby's randomized algorithm [Luby 1985].
18 *
19 * Given a numeric n x n adjacency matrix A of an unweighted and undirected graph (where
20 * the value true represents an edge), compute a maximal set of independent vertices and
21 * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
22 * of the set (the iset vector should be uninitialized on input.)
23 */
24 GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
25 {
26     GrB_Index n;
27     GrB_Matrix_nrows(&n,A); // n = # of rows of A
28
29     GrB_Vector prob; // holds random probabilities for each node
30     GrB_Vector neighbor_max; // holds value of max neighbor probability
31     GrB_Vector new_members; // holds set of new members to iset
32     GrB_Vector new_neighbors; // holds set of new neighbors to new iset mbrs.
33     GrB_Vector candidates; // candidate members to iset
34
35     GrB_Vector_new(&prob, GrB_FP32, n);
36     GrB_Vector_new(&neighbor_max, GrB_FP32, n);
37     GrB_Vector_new(&new_members, GrB_BOOL, n);
38     GrB_Vector_new(&new_neighbors, GrB_BOOL, n);
39     GrB_Vector_new(&candidates, GrB_BOOL, n);
40     GrB_Vector_new(iset, GrB_BOOL, n); // Initialize independent set vector, bool
41
42     GrB_Monoid Max;
43     GrB_Monoid_new(&Max, GrB_MAX_FP32, 0.0f);
44
45     GrB_Semiring maxSelect2nd; // Max/Select2nd "semiring"
46     GrB_Semiring_new(&maxSelect2nd, Max, GrB_SECOND_FP32);
47
48     GrB_Monoid Lor;
49     GrB_Monoid_new(&Lor, GrB_LOR, (bool) false);
50
51     GrB_Semiring Boolean; // Boolean semiring
52     GrB_Semiring_new(&Boolean, Lor, GrB_LAND);
53
54     // replace
55     GrB_Descriptor r_desc;
56     GrB_Descriptor_new(&r_desc);
57     GrB_Descriptor_set(r_desc, GrB_OUTP, GrB_REPLACE);
58
59     // replace + structural complement of mask
60     GrB_Descriptor sr_desc;
61     GrB_Descriptor_new(&sr_desc);
62     GrB_Descriptor_set(sr_desc, GrB_MASK, GrB_SCMP);

```

```

63 GrB_Descriptor_set(sr_desc, GrB_OUTP, GrB_REPLACE);
64
65 GrB_UnaryOp set_random;
66 GrB_UnaryOp_new(&set_random, setRandom, GrB_FP32, GrB_UINT32);
67
68 // compute the degree of each vertex.
69 GrB_Vector degrees;
70 GrB_Vector_new(&degrees, GrB_FP64, n);
71 GrB_reduce(degrees, GrB_NULL, GrB_NULL, GrB_PLUS_FP64, A, GrB_NULL);
72
73 // Isolated vertices are not candidates: candidates[degrees != 0] = true
74 GrB_assign(candidates, degrees, GrB_NULL, true, GrB_ALL, n, GrB_NULL);
75
76 // add all singletons to iset: iset[degree == 0] = 1
77 GrB_assign(*iset, degrees, GrB_NULL, true, GrB_ALL, n, sr_desc);
78
79 // Iterate while there are candidates to check.
80 GrB_Index nvals;
81 GrB_Vector_nvals(&nvals, candidates);
82 while (nvals > 0) {
83     // compute a random probability scaled by inverse of degree
84     GrB_apply(prob, candidates, GrB_NULL, set_random, degrees, r_desc);
85
86     // compute the max probability of all neighbors
87     GrB_mxr(neighbor_max, candidates, GrB_NULL, maxSelect2nd, A, prob, r_desc);
88
89     // select vertex if its probability is larger than all its active neighbors,
90     // and apply a "masked no-op" to remove stored falses
91     GrB_eWiseAdd(new_members, GrB_NULL, GrB_NULL, GrB_GT_FP64, prob, neighbor_max, GrB_NULL);
92     GrB_apply(new_members, new_members, GrB_NULL, GrB_IDENTITY_BOOL, new_members, r_desc);
93
94     // add new members to independent set.
95     GrB_eWiseAdd(*iset, GrB_NULL, GrB_NULL, GrB_LOR, *iset, new_members, GrB_NULL);
96
97     // remove new members from set of candidates  $c = c \& \text{!new}$ 
98     GrB_eWiseMult(candidates, new_members, GrB_NULL,
99                 GrB_LAND, candidates, candidates, sr_desc);
100
101     GrB_Vector_nvals(&nvals, candidates);
102     if (nvals == 0) { break; } // early exit condition
103
104     // Neighbors of new members can also be removed from candidates
105     GrB_mxr(new_neighbors, candidates, GrB_NULL, Boolean, A, new_members, GrB_NULL);
106     GrB_eWiseMult(candidates, new_neighbors, GrB_NULL,
107                 GrB_LAND, candidates, candidates, sr_desc);
108
109     GrB_Vector_nvals(&nvals, candidates);
110 }
111
112 GrB_free(&neighbor_max); // free all objects "new'ed"
113 GrB_free(&new_members);
114 GrB_free(&new_neighbors);
115 GrB_free(&prob);
116 GrB_free(&candidates);
117 GrB_free(&maxSelect2nd);
118 GrB_free(&Boolean);
119 GrB_free(&Max);
120 GrB_free(&Lor);
121 GrB_free(&sr_desc);
122 GrB_free(&r_desc);
123 GrB_free(&set_random);
124 GrB_free(&degrees);
125
126 return GrB_SUCCESS;
127 }

```

B.6 Example: counting triangles in GraphBLAS

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given, L, the lower triangular portion of n x n adjacency matrix A (of and
9  * undirected graph), computes the number of triangles in the graph.
10 */
11 uint64_t triangle_count(GrB_Matrix L)           // L: NxN, lower-triangular, bool
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, L);                   // n = # of vertices
15
16     GrB_Matrix C;
17     GrB_Matrix_new(&C, GrB_UINT64, n, n);
18
19     GrB_Monoid UInt64Plus;                      // integer plus monoid
20     GrB_Monoid_new(&UInt64Plus, GrB_PLUS_UINT64, 0, 1);
21
22     GrB_Semiring UInt64Arithmetic;             // integer arithmetic semiring
23     GrB_Semiring_new(&UInt64Arithmetic, UInt64Plus, GrB_TIMES_UINT64);
24
25     GrB_Descriptor desc_tb;                    // Descriptor for mm
26     GrB_Descriptor_new(&desc_tb);
27     GrB_Descriptor_set(desc_tb, GrB_INP1, GrB_TRAN); // transpose the second matrix
28
29     GrB_mxm(C, L, GrB_NULL, UInt64Arithmetic, L, L, desc_tb); // C<L> = L *.+ L'
30
31     uint64_t count;
32     GrB_reduce(&count, GrB_NULL, UInt64Plus, C, GrB_NULL); // 1-norm of C
33
34     GrB_free(&C);                               // C matrix no longer needed
35     GrB_free(&UInt64Arithmetic);               // Semiring no longer needed
36     GrB_free(&UInt64Plus);                     // Monoid no longer needed
37     GrB_free(&desc_tb);                        // descriptor no longer needed
38
39     return count;
40 }
```