

1 The GraphBLAS C API Specification †:

2 Version 1.3.0

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

4 Generated on 2019/09/25 at 15:32:56 EDT

†Based on *GraphBLAS Mathematics* by Jeremy Kepner

5 Copyright © 2017-2019 Carnegie Mellon University, The Regents of the University of California,
6 through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from
7 the U.S. Dept. of Energy), the Regents of the University of California (U.C. Davis), Intel Corpo-
8 ration, International Business Machines Corporation, and Massachusetts Institute of Technology
9 Lincoln 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 Except as otherwise noted, this material is licensed under a Creative Commons Attribution 4.0
22 license (<http://creativecommons.org/licenses/by/4.0/legalcode>), and examples are licensed under
23 the BSD License (<https://opensource.org/licenses/BSD-3-Clause>).

24 Contents

25	List of Tables	8
26	List of Figures	11
27	Acknowledgments	12
28	1 Introduction	13
29	2 Basic Concepts	15
30	2.1 Glossary	15
31	2.1.1 GraphBLAS API basic definitions	15
32	2.1.2 GraphBLAS objects and their structure	16
33	2.1.3 Algebraic structures used in the GraphBLAS	17
34	2.1.4 The execution of an application using the GraphBLAS C API	18
35	2.1.5 GraphBLAS methods: behaviors and error conditions	19
36	2.2 Notation	20
37	2.3 Algebraic and Arithmetic Foundations	21
38	2.4 GraphBLAS Opaque Objects	21
39	2.5 Domains	23
40	2.6 Operators and Associated Functions	23
41	2.7 Indices, Index Arrays, and Scalar Arrays	24
42	2.8 Execution Model	29
43	2.8.1 Execution modes	30
44	2.8.2 Thread safety	31
45	2.9 Error Model	31
46	3 Objects	35

47	3.1	Operators	37
48	3.2	Monoids	37
49	3.3	Semirings	37
50	3.4	Vectors	38
51	3.5	Matrices	39
52	3.6	Masks	39
53	3.7	Descriptors	40
54	4	Methods	43
55	4.1	Context Methods	43
56	4.1.1	init: Initialize a GraphBLAS context	43
57	4.1.2	finalize: Finalize a GraphBLAS context	44
58	4.1.3	getVersion: Get the version number of the standard.	45
59	4.2	Object Methods	45
60	4.2.1	Algebra Methods	46
61	4.2.1.1	Type_new: Create a new GraphBLAS (user-defined) type	46
62	4.2.1.2	UnaryOp_new: Create a new GraphBLAS unary operator	47
63	4.2.1.3	BinaryOp_new: Create a new GraphBLAS binary operator	48
64	4.2.1.4	Monoid_new: Create new GraphBLAS monoid	50
65	4.2.1.5	Semiring_new: Create new GraphBLAS semiring	51
66	4.2.2	Vector Methods	52
67	4.2.2.1	Vector_new: Create new vector	52
68	4.2.2.2	Vector_dup: Create a copy of a GraphBLAS vector	53
69	4.2.2.3	Vector_resize: Resize a vector	54
70	4.2.2.4	Vector_clear: Clear a vector	55
71	4.2.2.5	Vector_size: Size of a vector	56
72	4.2.2.6	Vector_nvals: Number of stored elements in a vector	56
73	4.2.2.7	Vector_build: Store elements from tuples into a vector	57
74	4.2.2.8	Vector_setElement: Set a single element in a vector	59
75	4.2.2.9	Vector_removeElement: Remove an element from a vector	61
76	4.2.2.10	Vector_extractElement: Extract a single element from a vector.	62

77	4.2.2.11	Vector_extractTuples: Extract tuples from a vector	63
78	4.2.3	Matrix Methods	65
79	4.2.3.1	Matrix_new: Create new matrix	65
80	4.2.3.2	Matrix_dup: Create a copy of a GraphBLAS matrix	66
81	4.2.3.3	Matrix_resize: Resize a matrix	67
82	4.2.3.4	Matrix_clear: Clear a matrix	68
83	4.2.3.5	Matrix_nrows: Number of rows in a matrix	69
84	4.2.3.6	Matrix_ncols: Number of columns in a matrix	70
85	4.2.3.7	Matrix_nvals: Number of stored elements in a matrix	71
86	4.2.3.8	Matrix_build: Store elements from tuples into a matrix	72
87	4.2.3.9	Matrix_setElement: Set a single element in matrix	74
88	4.2.3.10	Matrix_removeElement: Remove an element from a matrix	75
89	4.2.3.11	Matrix_extractElement: Extract a single element from a matrix	76
90	4.2.3.12	Matrix_extractTuples: Extract tuples from a matrix	78
91	4.2.4	Descriptor Methods	80
92	4.2.4.1	Descriptor_new: Create new descriptor	80
93	4.2.4.2	Descriptor_set: Set content of descriptor	81
94	4.2.5	free method	82
95	4.3	GraphBLAS Operations	83
96	4.3.1	mxm: Matrix-matrix multiply	87
97	4.3.2	vxm: Vector-matrix multiply	92
98	4.3.3	mxv: Matrix-vector multiply	96
99	4.3.4	eWiseMult: Element-wise multiplication	100
100	4.3.4.1	eWiseMult: Vector variant	100
101	4.3.4.2	eWiseMult: Matrix variant	105
102	4.3.5	eWiseAdd: Element-wise addition	110
103	4.3.5.1	eWiseAdd: Vector variant	110
104	4.3.5.2	eWiseAdd: Matrix variant	115
105	4.3.6	extract: Selecting Sub-Graphs	121
106	4.3.6.1	extract: Standard vector variant	121

107	4.3.6.2	extract: Standard matrix variant	125
108	4.3.6.3	extract: Column (and row) variant	130
109	4.3.7	assign: Modifying Sub-Graphs	135
110	4.3.7.1	assign: Standard vector variant	135
111	4.3.7.2	assign: Standard matrix variant	140
112	4.3.7.3	assign: Column variant	146
113	4.3.7.4	assign: Row variant	151
114	4.3.7.5	assign: Constant vector variant	157
115	4.3.7.6	assign: Constant matrix variant	162
116	4.3.8	apply: Apply a function to the elements of an object	167
117	4.3.8.1	apply: Vector variant	167
118	4.3.8.2	apply: Matrix variant	171
119	4.3.8.3	apply: Vector-BinaryOp variants	176
120	4.3.8.4	apply: Matrix-BinaryOp variants	180
121	4.3.9	reduce: Perform a reduction across the elements of an object	186
122	4.3.9.1	reduce: Standard matrix to vector variant	186
123	4.3.9.2	reduce: Vector-scalar variant	190
124	4.3.9.3	reduce: Matrix-scalar variant	193
125	4.3.10	transpose: Transpose rows and columns of a matrix	195
126	4.3.11	kroncker: Kronecker product of two matrices	199
127	4.4	Sequence Termination	205
128	4.4.1	wait: Wait for pending operations to complete	205
129	4.4.1.1	wait: Waits until all pending operations complete variant	205
130	4.4.1.2	wait: Waits until pending operations on a specific object complete	
131		variant	206
132	4.4.2	error: Get an error message regarding internal errors	207
133	5	Nonpolymorphic Interface	209
134	A	Revision History	217
135	B	Examples	221

136	B.1 Example: level breadth-first search (BFS) in GraphBLAS	222
137	B.2 Example: level BFS in GraphBLAS using apply	223
138	B.3 Example: parent BFS in GraphBLAS	224
139	B.4 Example: betweenness centrality (BC) in GraphBLAS	225
140	B.5 Example: batched BC in GraphBLAS	227
141	B.6 Example: maximal independent set (MIS) in GraphBLAS	229
142	B.7 Example: counting triangles in GraphBLAS	231

List of Tables

144	2.1	GraphBLAS opaque objects and their types.	22
145	2.2	Predefined <code>GrB_Type</code> values, the corresponding C type (for scalar parameters), and	
146		domains for GraphBLAS.	23
147	2.3	Valid GraphBLAS domain suffixes and corresponding C types (for I and T in Ta-	
148		bles 2.4, 2.5, 2.6, and 2.7).	24
149	2.4	Predefined unary and binary operators for GraphBLAS in C. The T can be any	
150		suffix from Table 2.3, I can be any integer suffix from Table 2.3, and F can be any	
151		floating-point suffix from Table 2.3.	25
152	2.5	Predefined monoids for GraphBLAS in C. Maximum and minimum values for the	
153		various integral types are defined in <code>stdint.h</code> . Floating-point infinities are defined	
154		in <code>math.h</code> . The x in <code>UINTx</code> or <code>INTx</code> can be one of 8, 16, 32, or 64; whereas in <code>FPx</code> ,	
155		it can be 32 or 64.	26
156	2.6	Predefined true semirings where the additive identity is the multiplicative annihilator.	
157		The x in <code>UINTx</code> or <code>INTx</code> can be one of 8, 16, 32, or 64; whereas in <code>FPx</code> , it can	
158		be 32 or 64.	27
159	2.7	Other useful predefined semirings that don't have a multiplicative annihilator. The	
160		x in <code>UINTx</code> or <code>INTx</code> can be one of 8, 16, 32, or 64; whereas in <code>FPx</code> , it can be 32 or 64.	28
161	2.8	Error values returned by GraphBLAS methods.	33
162	3.1	Operator input for relevant GraphBLAS operations. The semiring <code>add</code> and <code>times</code>	
163		are shown if applicable.	36
164	3.2	Properties and recipes for building GraphBLAS algebraic objects: unary operator,	
165		binary operator, monoid, and semiring (composed of operations <code>add</code> and <code>times</code>).	
166		Note 1: The output domain of the semiring <code>times</code> must be same as the domain of	
167		the semiring <code>add</code> . This ensures three domains for a semiring rather than four.	36
168	3.3	Descriptors are GraphBLAS objects passed as arguments to <code>Graph_BLAS</code> operations	
169		to modify other GraphBLAS objects in the operation's argument list. A descriptor,	
170		<code>desc</code> , has one or more (<i>field</i> , <i>value</i>) pairs indicated as <code>desc[GrB_Desc.Field].GrB_Desc.Value</code> .	
171		In this table, we define all types and literals used with descriptors.	41

172	3.4	Pre-defined GraphBLAS descriptors. The list includes all possible descriptors, according to the current standard. Columns list the possible fields and entries list the value(s) associated with those fields for a given descriptor.	42
173			
174			
175	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in this specification. Input matrices A and 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, C . The mask controls which values resulting from the operation on the right-hand side are written into the output object (complement and structure flags are not shown). 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).	84
176			
177			
178			
179			
180			
181			
182			
183			
184			
185			
186	5.1	Long-name, nonpolymorphic form of GraphBLAS methods.	209
187	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	210
188	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	211
189	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	212
190	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	213
191	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	214
192	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	215
193	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued).	216

194 **List of Figures**

195 2.1 Signature of GrB_error() function. 32

196 3.1 Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist
197 of a conventional monoid with one domain for the addition function, and a binary
198 operator with three domains for the multiplication function. 38

199 4.1 Flowchart for the GraphBLAS operations. Although shown specifically for the mxm
200 operation, many elements are common to all operations: such as the “ACCUM” and
201 “MASK and REPLACE” blocks. The triple arrows (\Rightarrow) denote where “as if copy”
202 takes place (including both collections and descriptor settings). The bold, dotted
203 arrows indicate where casting may occur between different domains. 85

204 Acknowledgments

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

207 Those who served as C API Subcommittee members for GraphBLAS 1.0 through 1.3 are (in al-
208 phabetical order):

- 209 • Aydin Buluç (Lawrence Berkeley National Laboratory)
- 210 • Timothy G. Mattson (Intel Corporation)
- 211 • Scott McMillan (Software Engineering Institute at Carnegie Mellon University)
- 212 • José Moreira (IBM Corporation)
- 213 • Carl Yang (UC Davis)

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

- 215 • The Department of Energy Office of Advanced Scientific Computing Research under contract
216 number DE-AC02-05CH11231
- 217 • Intel Corporation
- 218 • Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon Univer-
219 sity for the operation of the Software Engineering Institute [DM-0003727, DM19-0929]
- 220 • International Business Machines Corporation
- 221 • Department of Defense under contract No. W911QX-12-C-0059, L-3 Data Tactics subcontract
222 SCT-14-004 with University of California, Davis

223 The following people provided valuable input and feedback during the development of the specifi-
224 cation (in alphabetical order): Hollen Barmer, Benjamin Brock, Tim Davis, Jeremy Kepner, Peter
225 Kogge, Manoj Kumar, Andrew Mellinger, Maxim Naumov, Nancy M. Ott, Ping Tak Peter Tang,
226 Michael Wolf, Albert-Jan Yzelman.

227 Chapter 1

228 Introduction

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

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

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

243 The remainder of this document is organized as follows:

- 244 • Chapter 2: Basic Concepts
- 245 • Chapter 3: Objects
- 246 • Chapter 4: Methods
- 247 • Chapter 5: Nonpolymorphic Interface
- 248 • Appendix A: Revision History
- 249 • Appendix B: Examples

250 Chapter 2

251 Basic Concepts

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

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

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

263 2.1 Glossary

264 2.1.1 GraphBLAS API basic definitions

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

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

279 2.1.2 GraphBLAS objects and their structure

- 280 • *GraphBLAS object*: An instance of a data type defined by the GraphBLAS C API that
281 is opaque and manipulated only through the API. There are three groups of GraphBLAS
282 objects: *algebraic objects* (operators, monoids and semirings), *collections* (vectors, matrices
283 and masks), and descriptors. Because the object is based on an opaque datatype, an im-
284 plementation of the GraphBLAS C API has the flexibility to optimize data structures for a
285 particular platform. GraphBLAS objects are often implemented as sparse data structures,
286 meaning only the subset of the elements that have non-zero values are stored.
- 287 • *handle*: A variable that uses one of the GraphBLAS opaque data types. The value of
288 this variable holds a reference to a GraphBLAS object but not the contents of the object
289 itself. Hence, assigning a value of one handle to another variable copies the reference to the
290 GraphBLAS object but not the contents of the object.
- 291 • *non-opaque datatype*: Any datatype that exposes its internal structure. This is contrasted
292 with an *opaque datatype* that hides its internal structure and can be manipulated only through
293 an API.
- 294 • *domain*: The set of valid values for the elements of a GraphBLAS object. Note that some
295 GraphBLAS objects involve functions that map values from one or more input domains onto
296 values in an output domain. These GraphBLAS objects would have multiple domains.
- 297 • *implied zero*: Any element that has a valid index (or indices) in a GraphBLAS vector or
298 matrix but is not explicitly identified in the list of elements of that vector or matrix. From a
299 mathematical perspective, an *implied zero* is treated as having the value of the zero element of
300 the relevant monoid or semiring. However, GraphBLAS operations are purposefully defined
301 using set notation in such a way that it makes it unnecessary to reason about implied zeros.
302 Therefore, this concept is not used in the definition of GraphBLAS methods and operators.
- 303 • *mask*: An internal GraphBLAS object used to control how values are stored in a method's
304 output object. The mask exists only inside a method; hence, it is called an *internal opaque*
305 *object*. A mask is formed from the elements of a collection object (vector or matrix) input as
306 a mask parameter to a method. There are two different operations for forming the internal
307 mask.

308 GraphBLAS allows two types of masks:

- 309 1. The default behavior is that an element of the mask exists for each element that exists
 310 in the input collection object when the value of that element cast to a Boolean type
 311 evaluates to `true`.
- 312 2. In the *structure only* case, masks have structure but no values. The input collection
 313 describes a structure whereby an element of the mask exists for each element of the
 314 input collection regardless of its value.
- 315 • *complement*: The *complement* of a GraphBLAS mask, M , is another mask, M' , where the
 316 elements of M' are those elements from M that *do not* exist.

317 2.1.3 Algebraic structures used in the GraphBLAS

- 318 • *GraphBLAS operators*: Binary or unary operators that act on elements of GraphBLAS ob-
 319 jects. *GraphBLAS operators* are used to express algebraic structures used in the GraphBLAS
 320 such as monoids and semirings. There are two types of *GraphBLAS operators*: (1) predefined
 321 operators found in Table 2.4 and (2) user-defined operators created using `GrB_UnaryOp_new()`
 322 or `GrB_BinaryOp_new()` (see Section 4.2.1).

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

335 No GraphBLAS method will imply a predefined order over any associative operators. Im-
 336 plementations of the GraphBLAS are encouraged to exploit associativity to optimize per-
 337 formance of any GraphBLAS method. This holds even if the definition of the graphBLAS
 338 method implies a fixed order for the associative operations.

- 339 • *monoid*: An algebraic structure consisting of a domain, an associative binary operator, and
 340 an identity corresponding to that operator. There are two types of *GraphBLAS monoids*: (1)
 341 predefined monoids found in Table 2.5 and (2) user-defined monoids created using `GrB_Monoid_new()`
 342 (see Section 4.2.1).
- 343 • *semiring*: An algebraic structure consisting of a set of allowed values (the *domain*), two
 344 commutative binary operators called addition and multiplication (where multiplication dis-
 345 tributes over addition), and identities over addition (0) and multiplication (1). The additive
 346 identity is an annihilator over multiplication. Note that a *GraphBLAS semiring* is allowed to
 347 diverge from the mathematically rigorous definition of a semiring since certain combinations
 348 of domains, operators, and identity elements are useful in graph algorithms even when they do

349 not strictly match the mathematical definition of a semiring. There are two types of *Graph-*
350 *BLAS semirings*: (1) predefined semirings found in Tables 2.6 and 2.7, and (2) user-defined
351 semirings created using `GrB_Semiring_new()` (see Section 4.2.1).

352 2.1.4 The execution of an application using the GraphBLAS C API

- 353 • *program order*: The order of the GraphBLAS methods as defined by the text of an application
354 program.
- 355 • *sequence*: A series of GraphBLAS method calls in program order. An implementation of the
356 GraphBLAS may reorder or even fuse GraphBLAS methods within a sequence as long as the
357 definitions of any GraphBLAS object that is later read by an application are not changed; by
358 “read” we mean that values are copied from an opaque GraphBLAS object into a non-opaque
359 object. A sequence begins when a thread calls the first method that creates or modifies a
360 GraphBLAS object, either (1) the first call in an application or (2) the first call following
361 termination of a prior sequence. In blocking mode, every GraphBLAS method call is its own
362 sequence. In nonblocking mode, a sequence can be terminated by a call to `GrB_finalize()`,
363 a call to `GrB_wait()`, or by a series of `GrB_wait(obj)` method calls to every object that is an
364 output in the sequence.
- 365 • *complete*: The state of a GraphBLAS object when the computations that implement the
366 mathematical definition of the object have finished and the values associated with that com-
367 putation touches that object in the program’s address space. A GraphBLAS object is fully
368 defined by the sequence of methods. The execution of a sequence may be deferred, however,
369 so at any point in an application, a GraphBLAS object may not be materialized. That is,
370 the values associated with a particular GraphBLAS object may not have been computed and
371 stored in memory. An object is complete when the sequence that defines the object’s value
372 terminates or when a `GrB_wait()` method is called with that object as an argument.
- 373 • *materialize*: Cause the values associated with that object to be resident in memory and
374 visible to an application. A GraphBLAS object has been *materialized* when the computa-
375 tions that implement the mathematical definition of the object are *complete*. A GraphBLAS
376 object that is never loaded into a non-opaque data structure may potentially never be ma-
377 terialized. This might happen, for example, if the operations associated with the object are
378 fused or otherwise changed by the runtime system that supports the implementation of the
379 GraphBLAS C API.
- 380 • *context*: An instance of the GraphBLAS C API implementation as seen by an application.
381 An application can have only one context between the start and end of the application. A
382 context begins with the first thread that calls `GrB_init()` and ends with the first thread to call
383 `GrB_finalize()`. It is an error for `GrB_init()` or `GrB_finalize()` to be called more than one time
384 within an application. The context is used to constrain the behavior of an instance of the
385 GraphBLAS C API implementation and support various execution strategies. Currently, the
386 only supported constraints on a context pertain to the mode of program execution.
- 387 • *mode*: Defines how a GraphBLAS sequence executes, and is associated with the *context* of a
388 GraphBLAS C API implementation. It is set by an application with its call to `GrB_init()` to one

389 of two possible states. In *blocking mode*, GraphBLAS methods return after the computations
390 complete and any output objects have been updated. In *nonblocking mode*, a method may
391 return once the arguments are tested as consistent with the method (i.e., there are no API
392 errors), and potentially before any computation has taken place.

393 2.1.5 GraphBLAS methods: behaviors and error conditions

- 394 • *implementation defined behavior*: Behavior that must be documented by the implementation
395 and is allowed to vary among different compliant implementations.
- 396 • *undefined behavior*: Behavior that is not specified by the GraphBLAS C API. A conforming
397 implementation is free to choose results delivered from a method whose behavior is undefined.
- 398 • *thread-safe routine*: A routine that performs its intended function even when executed
399 concurrently (i.e., by more than one thread).
- 400 • *shape compatible objects*: GraphBLAS objects (matrices and vectors) that are passed as
401 parameters to a GraphBLAS method and have the correct number of dimensions and sizes for
402 each dimension to satisfy the rules of the mathematical definition of the operation associated
403 with the method. This is also referred to as *dimension compatible*.
- 404 • *domain compatible*: Two domains for which values from one domain can be cast to values in
405 the other domain as per the rules of the C language. In particular, domains from Table 2.2
406 are all compatible with each other, and a domain from a user-defined type is only compatible
407 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS
408 method ends and the domain mismatch error GrB_DOMAIN_MISMATCH is returned.

2.2 Notation

Notation	Description
$D_{out}, D_{in}, D_{in_1}, D_{in_2}$	Refers to output and input domains of various GraphBLAS operators.
$\mathbf{D}_{out}(*), \mathbf{D}_{in}(*),$ $\mathbf{D}_{in_1}(*), \mathbf{D}_{in_2}(*)$	Evaluates to output and input domains of GraphBLAS operators (usually a unary or binary operator, or semiring).
$\mathbf{D}(*)$	Evaluates to the (only) domain of a GraphBLAS object (usually a monoid, vector, or matrix).
f	An arbitrary unary function, usually a component of a unary operator.
$\mathbf{f}(F_u)$	Evaluates to the unary function contained in the unary operator given as the argument.
\odot	An arbitrary binary function, usually a component of a binary operator.
$\odot(*)$	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 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.

411 2.3 Algebraic and Arithmetic Foundations

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

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

421 Second, the ability to change semirings impacts the meaning of the *implied zero* in a sparse repre-
422 sentation of a matrix. This element in real arithmetic is zero, which is the identity of the *addition*
423 operator and the annihilator of the *multiplication* operator. As the semiring changes, this implied
424 zero changes to the identity of the *addition* operator and the annihilator of the *multiplication* op-
425 erator for the new semiring. Nothing changes in the stored matrix, but the implied zeros within
426 the sparse matrix or vector change with respect to a particular operation. In all cases, the nature
427 of the implied zero does not matter since the GraphBLAS C API treats them as elements of the
428 matrix or vector that do not exist.

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

440 2.4 GraphBLAS Opaque Objects

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

447 A GraphBLAS opaque object is accessed through its handle. A handle is a variable that uses
448 one of the types from Table 2.1. An implementation of the GraphBLAS specification has a great
449 deal of flexibility in how these handles are implemented. All that is required is that the handle

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.

450 corresponds to a type defined in the C language that supports assignment and comparison for
 451 equality. The GraphBLAS specification defines a literal `GrB_INVALID_HANDLE` that is valid for
 452 each type. Using the logical equality operator from C, it must be possible to compare a handle to
 453 `GrB_INVALID_HANDLE` to verify that a handle is valid.

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

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

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

473 In addition to opaque objects manipulated through handles, the GraphBLAS C API defines an
 474 additional opaque object as an internal object; that is, the object is never exposed as a variable
 475 within an application. This opaque object is the mask used to control how computed values are
 476 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

477 2.5 Domains

478 GraphBLAS defines two kinds of collections: matrices and vectors. For any given collection, the
 479 elements of the collection belong to a *domain*, which is the set of valid values for the elements. In
 480 GraphBLAS, domains correspond to the valid values for types from the host language (in our case,
 481 the C programming language). For any variable or object V in GraphBLAS we denote as $\mathbf{D}(V)$
 482 the domain of V , that is, the set of possible values that elements of V can take.

483 The predefined types and corresponding domains used in the GraphBLAS C API are shown in
 484 Table 2.2. The Boolean type (`bool`) is defined in `stdbool.h`, the integral types (`int8_t`, `uint8_t`,
 485 `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`) are defined in `stdint.h`, and the
 486 floating-point types (`float`, `double`) are native to the language and in most cases defined by the
 487 IEEE-754 standard.

488 2.6 Operators and Associated Functions

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

495 Similar to GraphBLAS types with predefined types and user-defined types, GraphBLAS operators
 496 come in two types: (1) predefined operators found in Table 2.4 and (2) user-defined operators using
 497 `GrB_UnaryOp_new()` or `GrB_BinaryOp_new()` (see Section 4.2.1).

498 Likewise, a list of predefined monoids, true semirings and convenience semirings can be found in

Table 2.3: Valid GraphBLAS domain suffixes and corresponding C types (for I and T in Tables 2.4, 2.5, 2.6, and 2.7).

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

Tables 2.5, 2.6 and 2.7, respectively. Predefined monoids are named `GrB_op_MONOID_T`, where op is the name of the predefined GraphBLAS operator used as the associative binary operation of the monoid and T is the domain (type) of the monoid. Predefined semirings are named `GrB_add_mul_SEMIRING_T`, where add is the semiring additive operation, mul is the semiring multiplicative operation and T is the domain (type) of the semiring.

The multiplicative inverse (`GrB_MINV_F`) function is only defined for floating-point types ($F = \text{FP32}$ or FP64). The division (`GrB_DIV_T`) function is defined for all types, but only if $y \neq 0$ for integral types and $y \neq \text{false}$ for the Boolean type.

2.7 Indices, Index Arrays, and Scalar Arrays

In order to interface with third-party software (i.e., software other than an implementation of the GraphBLAS), operations such as `GrB_Matrix_build` (Section 4.2.3.8) and `GrB_Matrix_extractTuples` (Section 4.2.3.12) must specify how the data should be laid out in non-opaque data structures. To this end we explicitly define the types for indices and the arrays used by these operations.

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

```
typedef uint64_t GrB_Index;
```

An index array is a pointer to a set of `GrB_Index` values that are stored in a contiguous block of memory (i.e., `GrB_Index*`). Likewise, a scalar array is a pointer to a contiguous block of memory storing a number of scalar values as specified by the user. Some GraphBLAS operations (e.g., `GrB_assign`) include an input parameter with the type of an index array. This input index array selects a subset of elements from a GraphBLAS vector object to be used in the operation. In these cases, the literal `GrB_ALL` can be used in place of the index array input parameter to indicate that

Table 2.4: Predefined unary and binary operators for GraphBLAS in C. The T can be any suffix from Table 2.3, I can be any integer suffix from Table 2.3, and F can be any floating-point suffix from Table 2.3.

Operator type	GraphBLAS identifier	Domains	Description	
GrB_UnaryOp	GrB_IDENTITY_ T	$T \rightarrow T$	$f(x) = x,$	identity
GrB_UnaryOp	GrB_ABS_ T	$T \rightarrow T$	$f(x) = x ,$	absolute value
GrB_UnaryOp	GrB_AINV_ T	$T \rightarrow T$	$f(x) = -x,$	additive inverse
GrB_UnaryOp	GrB_MINV_ F	$F \rightarrow F$	$f(x) = \frac{1}{x},$	multiplicative inverse
GrB_UnaryOp	GrB_LNOT	$\text{bool} \rightarrow \text{bool}$	$f(x) = \neg x,$	logical inverse
GrB_UnaryOp	GrB_BNOT_ I	$I \rightarrow I$	$f(x) = \sim x,$	bitwise complement
GrB_BinaryOp	GrB_LOR	$\text{bool} \times \text{bool} \rightarrow \text{bool}$	$f(x, y) = x \vee y,$	logical OR
GrB_BinaryOp	GrB_LAND	$\text{bool} \times \text{bool} \rightarrow \text{bool}$	$f(x, y) = x \wedge y,$	logical AND
GrB_BinaryOp	GrB_LXOR	$\text{bool} \times \text{bool} \rightarrow \text{bool}$	$f(x, y) = x \oplus y,$	logical XOR
GrB_BinaryOp	GrB_LXNOR	$\text{bool} \times \text{bool} \rightarrow \text{bool}$	$f(x, y) = \overline{x \oplus y},$	logical XNOR
GrB_BinaryOp	GrB_BOR_ I	$I \times I \rightarrow I$	$f(x, y) = x y,$	bitwise OR
GrB_BinaryOp	GrB_BAND_ I	$I \times I \rightarrow I$	$f(x, y) = x \& y,$	bitwise AND
GrB_BinaryOp	GrB_BXOR_ I	$I \times I \rightarrow I$	$f(x, y) = x \hat{\ } y,$	bitwise XOR
GrB_BinaryOp	GrB_BXNOR_ I	$I \times I \rightarrow I$	$f(x, y) = \overline{x \hat{\ } y},$	bitwise XNOR
GrB_BinaryOp	GrB_EQ_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x == y)$	equal
GrB_BinaryOp	GrB_NE_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x \neq y)$	not equal
GrB_BinaryOp	GrB_GT_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x > y)$	greater than
GrB_BinaryOp	GrB_LT_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x < y)$	less than
GrB_BinaryOp	GrB_GE_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x \geq y)$	greater than or equal
GrB_BinaryOp	GrB_LE_ T	$T \times T \rightarrow \text{bool}$	$f(x, y) = (x \leq y)$	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

Table 2.5: Predefined monoids for GraphBLAS in C. Maximum and minimum values for the various integral types are defined in `stdint.h`. Floating-point infinities are defined in `math.h`. The x in `UINT x` or `INT x` can be one of 8, 16, 32, or 64; whereas in `FP x` , it can be 32 or 64.

GraphBLAS identifier	Domains, T ($T \times T \rightarrow T$)	Identity	Description
<code>GrB_PLUS_MONOID_T</code>	<code>UINTx</code> <code>INTx</code> <code>FPx</code>	0 0 0	addition
<code>GrB_TIMES_MONOID_T</code>	<code>UINTx</code> <code>INTx</code> <code>FPx</code>	1 1 1	multiplication
<code>GrB_MIN_MONOID_T</code>	<code>UINTx</code> <code>INTx</code> <code>FPx</code>	<code>UINTx_MAX</code> <code>INTx_MAX</code> <code>INFINITY</code>	minimum
<code>GrB_MAX_MONOID_T</code>	<code>UINTx</code> <code>INTx</code> <code>FPx</code>	0 <code>INTx_MIN</code> <code>-INFINITY</code>	maximum
<code>GrB_LOR_MONOID_BOOL</code>	<code>BOOL</code>	<code>false</code>	logical OR
<code>GrB_LAND_MONOID_BOOL</code>	<code>BOOL</code>	<code>true</code>	logical AND
<code>GrB_LXOR_MONOID_BOOL</code>	<code>BOOL</code>	<code>false</code>	logical XOR (not equal)
<code>GrB_LXNOR_MONOID_BOOL</code>	<code>BOOL</code>	<code>true</code>	logical XNOR (equal)

Table 2.6: Predefined true semirings where the additive identity is the multiplicative annihilator. The x in $\text{UINT}x$ or $\text{INT}x$ can be one of 8, 16, 32, or 64; whereas in $\text{FP}x$, it can be 32 or 64.

GraphBLAS identifier	Domains, T ($T \times T \rightarrow T$)	+ identity \times annihilator	Description
GrB_PLUS_TIMES_SEMIRING_ T	$\text{UINT}x$ $\text{INT}x$ $\text{FP}x$	0 0 0	arithmetic semiring
GrB_MIN_PLUS_SEMIRING_ T	$\text{UINT}x$ $\text{INT}x$ $\text{FP}x$	$\text{UINT}x_MAX$ $\text{INT}x_MAX$ INFINITY	min-plus semiring
GrB_MAX_PLUS_SEMIRING_ T	$\text{INT}x$ $\text{FP}x$	$\text{INT}x_MIN$ -INFINITY	max-plus semiring
GrB_MIN_TIMES_SEMIRING_ T	$\text{UINT}x$	$\text{UINT}x_MAX$	min-times semiring
GrB_MIN_MAX_SEMIRING_ T	$\text{UINT}x$ $\text{INT}x$ $\text{FP}x$	$\text{UINT}x_MAX$ $\text{INT}x_MAX$ INFINITY	min-max semiring
GrB_MAX_MIN_SEMIRING_ T	$\text{UINT}x$ $\text{INT}x$ $\text{FP}x$	0 $\text{INT}x_MIN$ -INFINITY	max-min semiring
GrB_MAX_TIMES_SEMIRING_ T	$\text{UINT}x$	0	max-times semiring
GrB_PLUS_MIN_SEMIRING_ T	$\text{UINT}x$	0	plus-min semiring
GrB_LOR_LAND_SEMIRING_BOOL	BOOL	false	Logical semiring
GrB_LAND_LOR_SEMIRING_BOOL	BOOL	true	"and-or" semiring
GrB_LXOR_LAND_SEMIRING_BOOL	BOOL	false	same as NEQ_LAND
GrB_LXNOR_LOR_SEMIRING_BOOL	BOOL	true	same as EQ_LOR

Table 2.7: Other useful predefined semirings that don't have a multiplicative annihilator. The x in $\text{UINT}x$ or $\text{INT}x$ can be one of 8, 16, 32, or 64; whereas in $\text{FP}x$, it can be 32 or 64.

GraphBLAS identifier	Domains, T ($T \times T \rightarrow T$)	+ identity	Description
<code>GrB_MAX_PLUS_SEMIRING_T</code>	$\text{UINT}x$	0	max-plus semiring
<code>GrB_MIN_TIMES_SEMIRING_T</code>	$\text{INT}x$	$\text{INT}x_MAX$	min-times semiring
	$\text{FP}x$	INFINITY	
<code>GrB_MAX_TIMES_SEMIRING_T</code>	$\text{INT}x$	$\text{INT}x_MIN$	max-times semiring
	$\text{FP}x$	$-\text{INFINITY}$	
<code>GrB_PLUS_MIN_SEMIRING_T</code>	$\text{INT}x$	0	plus-min semiring
	$\text{FP}x$	0	
<code>GrB_MIN_FIRST_SEMIRING_T</code>	$\text{UINT}x$	$\text{UINT}x_MAX$	min-select first semiring
	$\text{INT}x$	$\text{INT}x_MAX$	
	$\text{FP}x$	INFINITY	
<code>GrB_MIN_SECOND_SEMIRING_T</code>	$\text{UINT}x$	$\text{UINT}x_MAX$	min-select second semiring
	$\text{INT}x$	$\text{INT}x_MAX$	
	$\text{FP}x$	INFINITY	
<code>GrB_MAX_FIRST_SEMIRING_T</code>	$\text{UINT}x$	0	max-select first semiring
	$\text{INT}x$	$\text{INT}x_MIN$	
	$\text{FP}x$	$-\text{INFINITY}$	
<code>GrB_MAX_SECOND_SEMIRING_T</code>	$\text{UINT}x$	0	max-select second semiring
	$\text{INT}x$	$\text{INT}x_MIN$	
	$\text{FP}x$	$-\text{INFINITY}$	

520 all indices of the associated GraphBLAS vector object should be used. As with any literal defined
521 in the GraphBLAS, an implementation of the GraphBLAS C API has considerable freedom in
522 terms of how `GrB_ALL` is defined. Since `GrB_ALL` is used as an argument for an array parameter, it
523 must use a type consistent with a pointer. `GrB_ALL` must also have a non-null value to distinguish
524 it from the erroneous case of passing a `NULL` pointer as an array.

525 2.8 Execution Model

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

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

536 The GraphBLAS method calls in program order are organized into contiguous and nonoverlapping
537 *sequences*. A sequence is an ordered collection of method calls as encountered by an executing
538 thread. (For more on threads and GraphBLAS, see Section 2.8.2.) A sequence begins with either
539 (1) the first GraphBLAS method called by a thread, or (2) the first method called by a thread
540 after the end of the previous sequence. A sequence can end (terminate) in a variety of ways. A call
541 to the GraphBLAS `GrB_wait()` method (Section 4.4.1.1) always ends a sequence. The GraphBLAS
542 `GrB_finalize()` method (Section 4.1.2) also implicitly ends a sequence. Finally, in blocking mode (see
543 below), each GraphBLAS method starts and ends its own sequence.

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

552 The opaqueness of GraphBLAS objects allows execution to proceed from one method to the next
553 even when operations are not complete. Processing of nonopaque objects is never deferred in Graph-
554 BLAS. That is, methods that consume nonopaque objects (e.g., `GrB_Matrix_build`, Section 4.2.3.8())
555 and methods that produce nonopaque objects (e.g., `GrB_Matrix_extractTuples()`, Section 4.2.3.12)
556 always finish consuming or producing those nonopaque objects before returning.

557 2.8.1 Execution modes

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

- 560 • *blocking*: In blocking mode, each method completes the GraphBLAS operation defined by
561 the method before proceeding to the next statement in program order. Output GraphBLAS
562 objects defined by a method are fully produced and stored in memory (i.e., they are *mate-*
563 *rialized*). In other words, it is as if each method call is its own sequence. Even mechanisms
564 that break the opaqueness of the GraphBLAS objects (e.g., performance monitors, debuggers,
565 memory dumps) will observe the operation as complete.
- 566 • *nonblocking*: In nonblocking mode, each method may return once the input arguments have
567 been inspected and verified to define a well formed GraphBLAS operation. (That is, there
568 are no API errors; see Section 2.9.) The GraphBLAS operation may not have completed, but
569 the output object is ready to be used by the next GraphBLAS method call. Completion of
570 *all* operations in a sequence, including any that may generate execution errors, is guaranteed
571 once the sequence terminates. Sequence termination is accomplished by a call to `GrB_wait()`.

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

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

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

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

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

- 595 • `GrB_BLOCKING` specifies the blocking mode context.

596 • GrB_NONBLOCKING specifies the nonblocking mode context.

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

600 2.8.2 Thread safety

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

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

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

616 2.9 Error Model

617 All GraphBLAS methods return a value of type GrB_Info to provide information available to the
618 system at the time the method returns. The returned value can be either GrB_SUCCESS or one of the
619 defined error values shown in Table 2.8. The errors fall into two groups: API errors (Table 2.8(a))
620 and execution errors (Table 2.8(b)).

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

629 Execution errors indicate that something went wrong during the execution of a legal GraphBLAS
630 method invocation. Their occurrence may depend on specifics of the executing environment and
631 data values being manipulated. This does not mean that execution errors are the fault of the

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

Figure 2.1: Signature of `GrB_error()` function.

632 GraphBLAS implementation. For example, a memory leak could arise from an applica-
633 tion’s source code (a “program error”), but it may manifest itself in different points of a program’s
634 execution (or not at all) depending on the platform, problem size, or what else is running at that
635 time. Index-out-of-bounds and insufficient space execution errors always indicate a program error.

636 In blocking mode, where each method executes to completion, a returned execution error value
637 applies to the specific method. If a GraphBLAS method, executing in blocking mode, returns with
638 any execution error from Table 2.8(b) other than `GrB_PANIC`, it is guaranteed that no argument
639 used as input-only has been modified. Output arguments may be left in an invalid state, and their
640 use downstream in the program flow may cause additional errors. If a GraphBLAS method returns
641 with a `GrB_PANIC` execution error, no guarantees can be made about the state of any program
642 data.

643 In nonblocking mode, execution errors can be deferred. A return value of `GrB_SUCCESS` only
644 guarantees that there are no API errors in the method invocation. If an execution error value is
645 returned by a method in nonblocking mode, it indicates that an error was found during execution
646 of the sequence, up to and including the `GrB_wait()` method (Section 4.4.1.1) call that ends the
647 sequence. When possible, that return value will provide information concerning the cause of the
648 error.

649 As discussed in Section 4.4.1.2, a `GrB_wait(obj)` on a specific GraphBLAS object `obj` does not
650 necessarily end a sequence. However, no additional errors on the methods of the sequence that
651 have `obj` as an `OUT` or `INOUT` argument can be reported. From a GraphBLAS perspective, those
652 methods are *complete*.

653 If a GraphBLAS method, executing in nonblocking mode, returns with any execution error from
654 Table 2.8(b) other than `GrB_PANIC`, it is guaranteed that no argument used as input-only through
655 the entire sequence has been modified. Any output argument in the sequence may be left in
656 an invalid state and its use downstream in the program flow may cause additional errors. If a
657 GraphBLAS method returns with a `GrB_PANIC`, no guarantees can be made about the state of any
658 program data.

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

Table 2.8: 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.

Every GraphBLAS object has a *lifetime*, which consists of the sequence of instructions executed in program order between the *creation* and the *destruction* of the object. Pre-defined objects (types, operators, monoids, semirings and descriptors) are created when the GraphBLAS context is initialized by a call to `GrB_init` and are destroyed when the GraphBLAS context is terminated by a call to `GrB_finalize`.

Additional objects can be created by a call to a *constructor*. Each kind of object has its own explicit constructor method: `GrB_Type_new`, `GrB_UnaryOp_new`, `GrB_BinaryOp_new`, `GrB_Monoid_new`, `GrB_Semiring_new`, `GrB_Descriptor_new`, `GrB_Vector_new`, `GrB_Matrix_new`. Furthermore, vectors and matrices can be constructed by duplicating another vector or matrix through calls to the methods `GrB_Vector_dup` and `GrB_Matrix_dup`, respectively. Objects explicitly created by a call to a constructor can be destroyed by a call to `GrB_free`. The behavior of a program that calls `GrB_free` on a pre-defined object is undefined.

Several GraphBLAS constructor methods take objects as input arguments and use these objects to create a new object. For all `GrB_*_new` methods, the lifetime of the created object must end strictly before the lifetime of any input objects. For example, a vector constructor `GrB_Vector_new` takes a type object as input. That type object must not be destroyed until after the created vector is destroyed. Similarly, a `GrB_Semiring_new` method takes a monoid and a binary operator as inputs. Neither of these can be destroyed until after the created semiring is destroyed.

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
kroncker	binary operator monoid semiring
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

698 The `GrB_Vector_dup` and `GrB_Matrix_dup` constructor methods behave differently. In these cases,
699 the input vector or matrix can be destroyed as soon as the call returns. However, the original type
700 object used to create the input vector or matrix cannot be destroyed until after the vector or matrix
701 created by `GrB_Vector_dup` or `GrB_Matrix_dup` is destroyed. This behavior must hold for any chain
702 of duplicating constructors.

703 3.1 Operators

704 A GraphBLAS *binary operator* $F_b = \langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$ is defined by three domains, D_{out} ,
705 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
706 $F_b = \langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$, 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
707 $\odot(F_b) = \odot$. Note that \odot could be used in place of either \oplus or \otimes in other methods and operations.

708 A GraphBLAS *unary operator* $F_u = \langle D_{out}, D_{in}, f \rangle$ is defined by two domains, D_{out} and D_{in} , and
709 an operation $f : D_{in} \rightarrow D_{out}$. For a given GraphBLAS operator $F_u = \langle D_{out}, D_{in}, f \rangle$, we define
710 $\mathbf{D}_{out}(F_u) = D_{out}$, $\mathbf{D}_{in}(F_u) = D_{in}$, and $\mathbf{f}(F_u) = f$.

711 3.2 Monoids

712 A GraphBLAS *monoid* $M = \langle D, \odot, 0 \rangle$ is defined by a single domain D , an *associative*¹ operation
713 $\odot : D \times D \rightarrow D$, and an identity element $0 \in D$. For a given GraphBLAS monoid $M = \langle D, \odot, 0 \rangle$
714 we define $\mathbf{D}(M) = D$, $\odot(M) = \odot$, and $\mathbf{0}(M) = 0$. A GraphBLAS monoid is equivalent to the
715 conventional *monoid* algebraic structure.

716 Let $F = \langle D, D, D, \odot \rangle$ be an associative GraphBLAS binary operator with identity element $0 \in D$.
717 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
718 a *commutative monoid*. If a monoid M is created using an operator \odot that is not associative, the
719 outcome of GraphBLAS operations using such a monoid is undefined.

720 3.3 Semirings

721 A GraphBLAS *semiring* $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$ is defined by three domains D_{out} , D_{in_1} , and
722 D_{in_2} ; an *associative*¹ and commutative additive operation $\oplus : D_{out} \times D_{out} \rightarrow D_{out}$; a multiplicative
723 operation $\otimes : D_{in_1} \times D_{in_2} \rightarrow D_{out}$; and an identity element $0 \in D_{out}$. For a given GraphBLAS
724 semiring $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$ we define $\mathbf{D}_{in_1}(S) = D_{in_1}$, $\mathbf{D}_{in_2}(S) = D_{in_2}$, $\mathbf{D}_{out}(S) =$
725 D_{out} , $\oplus(S) = \oplus$, $\otimes(S) = \otimes$, and $\mathbf{0}(S) = 0$.

726 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,
727 then $S = \langle A, F \rangle = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$ is a semiring.

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.

728 In a GraphBLAS semiring, the multiplicative operator does not have to distribute over the additive
 729 operator. This is unlike the conventional *semiring* algebraic structure.

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

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

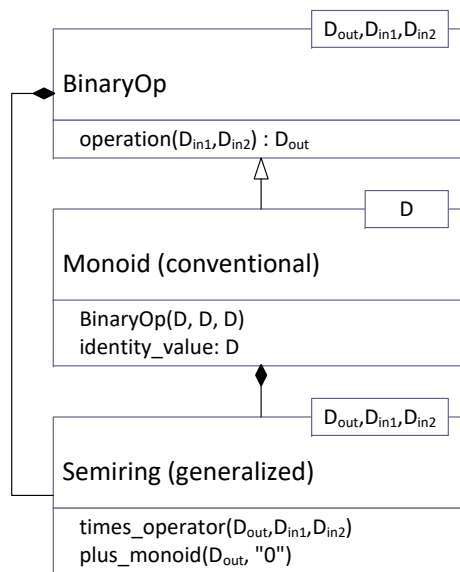


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

736 3.4 Vectors

737 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)
 738 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
 739 $\mathbf{size}(\mathbf{v}) = N$ 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
 740 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} ,
 741 $\mathbf{v}(i)$ is a reference to v_i if $(i, v_i) \in \mathbf{L}(\mathbf{v})$ and is undefined otherwise.

742 3.5 Matrices

743 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
744 number of columns $N > 0$, and a set of tuples (i, j, A_{ij}) where $0 \leq i < M$, $0 \leq j < N$, and
745 $A_{ij} \in D$. A particular pair of values i, j can appear at most once in \mathbf{A} . We define $\mathbf{ncols}(\mathbf{A}) = N$,
746 $\mathbf{nrows}(\mathbf{A}) = M$, 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
747 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
748 are the sets of nonempty rows and columns of \mathbf{A} , respectively.) The *structure* of matrix \mathbf{A} is the
749 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
750 A_{ij} if $(i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})$ and is undefined otherwise.

751 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
752 vector called the j -th *column* of \mathbf{A} . Correspondingly, if \mathbf{A} is a matrix and $0 \leq i < M$, then
753 $\mathbf{A}(i, :) = \langle D, N, \{(j, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\} \rangle$ is a vector called the i -th *row* of \mathbf{A} .

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

756 3.6 Masks

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

760 The mask is formed from objects input to the method that uses the mask. For example, a Graph-
761 BLAS method may be called with a matrix as the mask parameter. The internal mask object is
762 constructed from the input matrix in one of two ways. In the default case, an element of the mask
763 is created for each tuple that exists in the matrix for which the value of the tuple cast to Boolean
764 evaluates to **true**. Alternatively, the user can specify *structure-only* behavior where an element of
765 the mask is created for each tuple that exists in the matrix *regardless* of the value stored in the
766 input matrix.

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

774 A one-dimensional mask $\mathbf{m} = \langle N, \{i\} \rangle$ is defined by its number of elements $N > 0$, and a set
775 $\mathbf{ind}(\mathbf{m})$ of indices $\{i\}$ where $0 \leq i < N$. A particular value of i can appear at most once in \mathbf{m} . We
776 define $\mathbf{size}(\mathbf{m}) = N$. The set $\mathbf{ind}(\mathbf{m})$ is called the *structure* of mask \mathbf{m} .

777 A two-dimensional mask $\mathbf{M} = \langle M, N, \{(i, j)\} \rangle$ is defined by its number of rows $M > 0$, its number
778 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
779 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

780 also define the sets $\mathbf{indrow}(\mathbf{M}) = \{i : \exists(i, j) \in \mathbf{ind}(\mathbf{M})\}$ and $\mathbf{indcol}(\mathbf{M}) = \{j : \exists(i, j) \in \mathbf{ind}(\mathbf{M})\}$.
 781 These are the sets of nonempty rows and columns of \mathbf{M} , respectively. The set $\mathbf{ind}(\mathbf{M})$ is called the
 782 *structure* of mask \mathbf{M} .

783 One common operation on masks is the *complement*. For a one-dimensional mask \mathbf{m} this is denoted
 784 as $\neg\mathbf{m}$. For a two-dimensional masks, this is denoted as $\neg\mathbf{M}$. The complement of a one-dimensional
 785 mask \mathbf{m} is defined as $\mathbf{ind}(\neg\mathbf{m}) = \{i : 0 \leq i < N, i \notin \mathbf{ind}(\mathbf{m})\}$. It is the set of all possible indices
 786 that do not appear in \mathbf{m} . The complement of a two-dimensional mask \mathbf{M} is defined as the set
 787 $\mathbf{ind}(\neg\mathbf{M}) = \{(i, j) : 0 \leq i < M, 0 \leq j < N, (i, j) \notin \mathbf{ind}(\mathbf{M})\}$. It is the set of all possible indices
 788 that do not appear in \mathbf{M} .

789 3.7 Descriptors

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

794 The descriptor is a lightweight object. It is composed of $(field, value)$ pairs where the *field* selects
 795 one of the GraphBLAS objects from the argument list of a method and the *value* defines the
 796 indicated modification associated with that object. For example, a descriptor may specify that a
 797 particular input matrix needs to be transposed or that a mask needs to be complemented (defined
 798 in Section 3.6) before using it in the operation.

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

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

- 814 • Input matrices are not transposed.
- 815 • The mask is used, as is, without complementing, and stored values are examined to determine
 816 whether they evaluate to `true` or `false`.
- 817 • Values of the output object that are not directly modified by the operation are preserved.

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_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the associated object. The stored values are not examined.
<code>GrB_COMP</code>	Use the complement of the associated object. When combined with <code>GrB_STRUCTURE</code> , the complement of the structure of the associated object is used without evaluating the values stored.
<code>GrB_SCMP</code>	Use the complement of the associated object. When combined with <code>GrB_STRUCTURE</code> , the complement of the structure of the associated object is used without evaluating the values stored. This field value is currently deprecated in favor of <code>GrB_COMP</code> above, and may be removed in future versions of this API.
<code>GrB_TRAN</code>	Use the transpose of the associated object.
<code>GrB_REPLACE</code>	Clear the output object before assigning computed values.

818 GraphBLAS specifies a set of pre-defined descriptors. Their identifiers and the corresponding set
 819 of (field,value) pairs for that identifier are shown in Table 3.4.

Table 3.4: Pre-defined GraphBLAS descriptors. The list includes all possible descriptors, according to the current standard. Columns list the possible fields and entries list the value(s) associated with those fields for a given descriptor.

Identifier	GrB_OUTP	GrB_MASK	GrB_INP0	GrB_INP1
GrB_NULL	–	–	–	–
GrB_DESC_T1	–	–	–	GrB_TRAN
GrB_DESC_T0	–	–	GrB_TRAN	–
GrB_DESC_T0T1	–	–	GrB_TRAN	GrB_TRAN
GrB_DESC_C	–	GrB_COMP	–	–
GrB_DESC_S	–	GrB_STRUCTURE	–	–
GrB_DESC_CT1	–	GrB_COMP	–	GrB_TRAN
GrB_DESC_ST1	–	GrB_STRUCTURE	–	GrB_TRAN
GrB_DESC_CT0	–	GrB_COMP	GrB_TRAN	–
GrB_DESC_ST0	–	GrB_STRUCTURE	GrB_TRAN	–
GrB_DESC_CT0T1	–	GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_ST0T1	–	GrB_STRUCTURE	GrB_TRAN	GrB_TRAN
GrB_DESC_SC	–	GrB_STRUCTURE, GrB_COMP	–	–
GrB_DESC_SCT1	–	GrB_STRUCTURE, GrB_COMP	–	GrB_TRAN
GrB_DESC_SCT0	–	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	–
GrB_DESC_SCT0T1	–	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_R	GrB_REPLACE	–	–	–
GrB_DESC_RT1	GrB_REPLACE	–	–	GrB_TRAN
GrB_DESC_RT0	GrB_REPLACE	–	GrB_TRAN	–
GrB_DESC_RT0T1	GrB_REPLACE	–	GrB_TRAN	GrB_TRAN
GrB_DESC_RC	GrB_REPLACE	GrB_COMP	–	–
GrB_DESC_RS	GrB_REPLACE	GrB_STRUCTURE	–	–
GrB_DESC_RCT1	GrB_REPLACE	GrB_COMP	–	GrB_TRAN
GrB_DESC_RST1	GrB_REPLACE	GrB_STRUCTURE	–	GrB_TRAN
GrB_DESC_RCT0	GrB_REPLACE	GrB_COMP	GrB_TRAN	–
GrB_DESC_RST0	GrB_REPLACE	GrB_STRUCTURE	GrB_TRAN	–
GrB_DESC_RCT0T1	GrB_REPLACE	GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_RST0T1	GrB_REPLACE	GrB_STRUCTURE	GrB_TRAN	GrB_TRAN
GrB_DESC_RSC	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	–	–
GrB_DESC_RSCT1	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	–	GrB_TRAN
GrB_DESC_RSCT0	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	–
GrB_DESC_RSCT0T1	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	GrB_TRAN

820 Chapter 4

821 Methods

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

824 We would like to emphasize that no GraphBLAS method will imply a predefined order over any
825 associative operators. Implementations of the GraphBLAS are encouraged to exploit associativity
826 to optimize performance of any GraphBLAS method. This holds even if the definition of the
827 GraphBLAS method implies a fixed order for the associative operations.

828 4.1 Context Methods

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

832 4.1.1 `init`: Initialize a GraphBLAS context

833 Creates and initializes a GraphBLAS C API context.

834 C Syntax

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

836 Parameters

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

838 **Return Values**

839 `GrB_SUCCESS` operation completed successfully.

840 `GrB_PANIC` unknown internal error.

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

842 **Description**

843 The `init` method creates and initializes a GraphBLAS C API context. The argument to `GrB_init`
844 defines the mode for the context. The two available modes are:

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

854 An application can only create one context per execution instance. An application may only call
855 `GrB_init` once. Calling `GrB_init` more than once results in undefined behavior.

856 **4.1.2 finalize: Finalize a GraphBLAS context**

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

858 **C Syntax**

```
859 GrB_Info GrB_finalize();
```

860 **Return Values**

861 `GrB_SUCCESS` operation completed successfully.

862 `GrB_PANIC` unknown internal error.

863 **Description**

864 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS
865 C API context. `GrB_finalize` may only be called after a context has been initialized by calling
866 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-
867 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined
868 behavior.

869 **4.1.3 getVersion: Get the version number of the standard.**

870 Query the library for the version number of the standard that this library implements.

871 **C Syntax**

```
872         GrB_Info GrB_getVersion(unsigned int *version,  
873                               unsigned int *subversion);
```

874 **Parameters**

875 `version` (OUT) On successful return will hold the value of the major version number.

876 `subversion` (OUT) On successful return will hold the value of the subversion number.

877 **Return Values**

878 `GrB_SUCCESS` operation completed successfully.

879 `GrB_PANIC` unknown internal error.

880 **Description**

881 The `getVersion` method is used to query the major and minor version number of the GraphBLAS
882 C API specification that the library implements at runtime. To support compile time queries the
883 following two macros shall also be defined by the library.

```
884         #define GRB_VERSION      1  
885         #define GrB_SUBVERSION  3
```

886 **4.2 Object Methods**

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

889 4.2.1 Algebra Methods

890 4.2.1.1 Type_new: Create a new GraphBLAS (user-defined) type

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

893 C Syntax

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

896 Parameters

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

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

900 Return Values

901 GrB_SUCCESS operation completed successfully.

902 GrB_PANIC unknown internal error.

903 GrB_OUT_OF_MEMORY not enough memory available for operation.

904 GrB_NULL_POINTER utype pointer is NULL.

905 Description

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

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

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

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

915 **4.2.1.2 UnaryOp_new: Create a new GraphBLAS unary operator**

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

918 **C Syntax**

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

923 **Parameters**

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

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

```
929         void func(void *out, const void *in);  
930
```

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

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

937 **Return Values**

938 GrB_SUCCESS operation completed successfully.

939 GrB_PANIC unknown internal error.

940 GrB_OUT_OF_MEMORY not enough memory available for operation.

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

943 GrB_NULL_POINTER unary_op or unary_func pointers are NULL.

944 **Description**

945 The `UnaryOp_new` method creates a new GraphBLAS unary operator $f_u = \langle \mathbf{D}(d_out), \mathbf{D}(d_in), \text{unary_func} \rangle$
946 and returns a handle to it in `unary_op`.

947 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments
948 are aliased. In other words, for all invocations of the function:

```
949     unary_func(out, in);
```

950 the value of `out` must be the same as if the following code was executed:

```
951     D(d_in) tmp = malloc(sizeof(D(d_in)));  
952     memcpy(tmp, in, sizeof(D(d_in)));  
953     unary_func(out, tmp);  
954     free(tmp);
```

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

957 **4.2.1.3 BinaryOp_new: Create a new GraphBLAS binary operator**

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

960 **C Syntax**

```
961     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
962                             void          (*binary_func)(void*,  
963                                                         const void*,  
964                                                         const void*),  
965                             GrB_Type      d_out,  
966                             GrB_Type      d_in1,  
967                             GrB_Type      d_in2);
```

968 **Parameters**

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

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

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

975

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

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

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

985 Return Values

986 `GrB_SUCCESS` operation completed successfully.

987 `GrB_PANIC` unknown internal error.

988 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

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

991 `GrB_NULL_POINTER` `binary_op` or `binary_func` pointer is `NULL`.

992 Description

993 The `BinaryOp_new` methods creates a new GraphBLAS binary operator $f_b = \langle \mathbf{D}(d_out), \mathbf{D}(d_in1), \mathbf{D}(d_in2), \text{binary_func} \rangle$
994 and returns a handle to it in `binary_op`.

995 The implementation of `binary_func` must be such that it works even if any of the `d_out`, `d_in1`, and
996 `d_in2` arguments are aliased to each other. In other words, for all invocations of the function:

997 `binary_func(out, in1, in2);`

998 the value of `out` must be the same as if the following code was executed:

```
999 D(d_in1) tmp1 = malloc(sizeof(D(d_in1)));  
1000 D(d_in2) tmp2 = malloc(sizeof(D(d_in2)));  
1001 memcpy(tmp1, in1, sizeof(D(d_in1)));  
1002 memcpy(tmp2, in2, sizeof(D(d_in2)));  
1003 binary_func(out, tmp1, tmp2);  
1004 free(tmp2);  
1005 free(tmp1);
```

1006 It is not an error to call this method more than once on the same variable; however, the handle to
1007 the previously created object will be overwritten.

1008 **4.2.1.4 Monoid_new: Create new GraphBLAS monoid**

1009 Creates a new monoid with specified binary operator and identity value.

1010 **C Syntax**

```
1011         GrB_Info GrB_Monoid_new(GrB_Monoid  *monoid,  
1012                               GrB_BinaryOp  binary_op,  
1013                               <type>        identity);
```

1014 **Parameters**

1015 **monoid** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1016 monoid object.

1017 **binary_op** (IN) An existing GraphBLAS associative binary operator whose input and output
1018 types are the same.

1019 **identity** (IN) The value of the identity element of the monoid. Must be the same type as
1020 the type used by the **binary_op** operator.

1021 **Return Values**

1022 **GrB_SUCCESS** operation completed successfully.

1023 **GrB_PANIC** unknown internal error.

1024 **GrB_OUT_OF_MEMORY** not enough memory available for operation.

1025 **GrB_UNINITIALIZED_OBJECT** the **GrB_BinaryOp** has not been initialized by a call to **GrB_BinaryOp_new**.

1026 **GrB_NULL_POINTER** monoid pointer is NULL.

1027 **GrB_DOMAIN_MISMATCH** all three argument types of the binary operator and the type of the
1028 identity value are not the same.

1029 **Description**

1030 The **Monoid_new** method creates a new monoid $M = \langle \mathbf{D}(\text{binary_op}), \text{binary_op}, \text{identity} \rangle$ and returns
1031 a handle to it in **monoid**.

1032 If **binary_op** is not associative, the results of GraphBLAS operations that require associativity of
1033 this monoid will be undefined.

1034 It is not an error to call this method more than once on the same variable; however, the handle to
1035 the previously created object will be overwritten.

1036 **4.2.1.5 Semiring_new: Create new GraphBLAS semiring**

1037 Creates a new semiring with specified domain, operators, and elements.

1038 **C Syntax**

```
1039         GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1040                                 GrB_Monoid    add_op,  
1041                                 GrB_BinaryOp   mul_op);
```

1042 **Parameters**

1043 **semiring** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1044 semiring.

1045 **add_op** (IN) An existing GraphBLAS commutative monoid that specifies the addition op-
1046 erator and its identity.

1047 **mul_op** (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-
1048 plication operator. In addition, **mul_op**'s output domain, $\mathbf{D}_{out}(\mathbf{mul_op})$, must be
1049 the same as the **add_op**'s domain $\mathbf{D}(\mathbf{add_op})$.

1050 **Return Values**

1051 **GrB_SUCCESS** operation completed successfully.

1052 **GrB_PANIC** unknown internal error.

1053 **GrB_OUT_OF_MEMORY** not enough memory available for this method to complete.

1054 **GrB_UNINITIALIZED_OBJECT** the **add_op** object has not been initialized with a call to **GrB_Monoid_new**
1055 or the **mul_op** object has not been not been initialized by a call to
1056 **GrB_BinaryOp_new**.

1057 **GrB_NULL_POINTER** semiring pointer is NULL.

1058 **GrB_DOMAIN_MISMATCH** the output domain of **mul_op** does not match the domain of the
1059 **add_op** monoid.

1060 Description

1061 The `Semiring_new` method creates a new semiring $S = \langle \mathbf{D}_{out}(\text{mul_op}), \mathbf{D}_{in_1}(\text{mul_op}), \mathbf{D}_{in_2}(\text{mul_op}), \text{add_op}, \text{mul_op}, 0 \rangle$
1062 and returns a handle to it in `semiring`. Note that $\mathbf{D}_{out}(\text{mul_op})$ must be the same as $\mathbf{D}(\text{add_op})$.

1063 If `add_op` is not commutative, then GraphBLAS operations using this semiring will be undefined.

1064 It is not an error to call this method more than once on the same variable; however, the handle to
1065 the previously created object will be overwritten.

1066 4.2.2 Vector Methods

1067 4.2.2.1 Vector_new: Create new vector

1068 Creates a new vector with specified domain and size.

1069 C Syntax

```
1070     GrB_Info GrB_Vector_new(GrB_Vector *v,  
1071                           GrB_Type   d,  
1072                           GrB_Index  nsize);
```

1073 Parameters

1074 `v` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1075 vector.

1076 `d` (IN) The type corresponding to the domain of the vector being created. Can be
1077 one of the predefined GraphBLAS types in Table 2.2, or an existing user-defined
1078 GraphBLAS type.

1079 `nsize` (IN) The size of the vector being created.

1080 Return Values

1081 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1082 blocking mode, this indicates that the API checks for the input
1083 arguments passed successfully. Either way, output vector `v` is ready
1084 to be used in the next method of the sequence.

1085 `GrB_PANIC` Unknown internal error.

1086 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1087 GraphBLAS objects (input or output) is in an invalid state caused
1088 by a previous execution error. Call `GrB_error()` to access any error
1089 messages generated by the implementation.

1119 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1120 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, u , has not been initialized by a call to
1121 Vector_new or Vector_dup.

1122 GrB_NULL_POINTER The w pointer is NULL.

1123 **Description**

1124 Creates a new vector w of domain $D(u)$, size $size(u)$, and contents $L(u)$. The method returns a
1125 handle to the new vector in w .

1126 It is not an error to call this method more than once on the same variable; however, the handle to
1127 the previously created object will be overwritten.

1128 **4.2.2.3 Vector_resize: Resize a vector**

1129 Changes the size of an existing vector.

1130 **C Syntax**

```
1131           GrB_Info GrB_Vector_resize(GrB_Vector w,
1132                                    GrB_Index nsize);
```

1133 **Parameters**

1134 w (INOUT) An existing Vector object that is being resized.

1135 $nsize$ (IN) The new size of the vector. It can be smaller or larger than the current size.

1136 **Return Values**

1137 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1138 blocking mode, this indicates that the API checks for the input
1139 arguments passed successfully. Either way, output vector w is ready
1140 to be used in the next method of the sequence.

1141 GrB_PANIC Unknown internal error.

1142 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1143 GraphBLAS objects (input or output) is in an invalid state caused
1144 by a previous execution error. Call GrB_error() to access any error
1145 messages generated by the implementation.

1146 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1147 GrB_NULL_POINTER The w pointer is NULL.

1148 GrB_INVALID_VALUE $nsize$ is zero.

1149 **Description**

1150 Changes the size of w to $nsize$. The domain $\mathbf{D}(w)$ of vector w remains the same. The contents $\mathbf{L}(w)$
1151 are modified as described below.

1152 Let $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$ when the method is called. When the method returns, $w = \langle \mathbf{D}(w), nsize, \mathbf{L}'(w) \rangle$
1153 where $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < nsize)\}$. That is, all elements of w with index greater
1154 than or equal to the new vector size ($nsize$) are dropped.

1155 **4.2.2.4 Vector_clear: Clear a vector**

1156 Removes all the elements (tuples) from a vector.

1157 **C Syntax**

```
1158           GrB_Info GrB_Vector_clear(GrB_Vector v);
```

1159 **Parameters**

1160 v (INOUT) An existing GraphBLAS vector to clear.

1161 **Return Values**

1162 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1163 blocking mode, this indicates that the API checks for the input
1164 arguments passed successfully. Either way, output vector v is ready
1165 to be used in the next method of the sequence.

1166 GrB_PANIC Unknown internal error.

1167 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1168 GraphBLAS objects (input or output) is in an invalid state caused
1169 by a previous execution error. Call `GrB_error()` to access any error
1170 messages generated by the implementation.

1171 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1172 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v , has not been initialized by a call to
1173 `Vector_new` or `Vector_dup`.

1174 **Description**

1175 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`, $L(v) =$
1176 \emptyset . The size of the vector does not change.

1177 **4.2.2.5 Vector_size: Size of a vector**

1178 Retrieve the size of a vector.

1179 **C Syntax**

```
1180         GrB_Info GrB_Vector_size(GrB_Index      *nsize,  
1181                                const GrB_Vector v);
```

1182 **Parameters**

1183 nsize (OUT) On successful return, is set to the size of the vector.

1184 v (IN) An existing GraphBLAS vector being queried.

1185 **Return Values**

1186 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1187 cessfully and the value of nsize has been set.

1188 GrB_PANIC Unknown internal error.

1189 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1190 GraphBLAS objects (input or output) is in an invalid state caused
1191 by a previous execution error. Call `GrB_error()` to access any error
1192 messages generated by the implementation.

1193 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, v, has not been initialized by a call to
1194 `Vector_new` or `Vector_dup`.

1195 GrB_NULL_POINTER nsize pointer is NULL.

1196 **Description**

1197 Return `size(v)` in nsize.

1198 **4.2.2.6 Vector_nvals: Number of stored elements in a vector**

1199 Retrieve the number of stored elements (tuples) in a vector.

1200 **C Syntax**

```
1201         GrB_Info GrB_Vector_nvals(GrB_Index      *nvals,  
1202                                 const GrB_Vector v);
```

1203 **Parameters**

1204 **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)
1205 in the vector.

1206 **v** (IN) An existing GraphBLAS vector being queried.

1207 **Return Values**

1208 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1209 cessfully and the value of **nvals** has been set.

1210 **GrB_PANIC** Unknown internal error.

1211 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1212 GraphBLAS objects (input or output) is in an invalid state caused
1213 by a previous execution error. Call **GrB_error()** to access any error
1214 messages generated by the implementation.

1215 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1216 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to
1217 **Vector_new** or **Vector_dup**.

1218 **GrB_NULL_POINTER** The **nvals** pointer is **NULL**.

1219 **Description**

1220 Return **nvals(v)** in **nvals**. This is the number of stored elements in vector **v**, which is the size of
1221 **L(v)** (see Section 3.4).

1222 **4.2.2.7 Vector_build: Store elements from tuples into a vector**

1223 **C Syntax**

```
1224         GrB_Info GrB_Vector_build(GrB_Vector      w,  
1225                                 const GrB_Index  *indices,  
1226                                 const <type>     *values,  
1227                                 GrB_Index        n,  
1228                                 const GrB_BinaryOp dup);
```

1229 **Parameters**

- 1230 **w** (INOUT) An existing Vector object to store the result.
- 1231 **indices** (IN) Pointer to an array of indices.
- 1232 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of
1233 vector **w**.
- 1234 **n** (IN) The number of entries contained in each array (the same for indices and values).
- 1235 **dup** (IN) An associative and commutative binary operator to apply when duplicate
1236 values for the same location are present in the input arrays. All three domains of
1237 **dup** must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$.

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
1241 arguments passed successfully. Either way, output vector **w** 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** Either **w** has not been initialized by a call to `GrB_Vector_new` or
1250 by `GrB_Vector_dup`, or **dup** has not been initialized by a call to `GrB_BinaryOp_new`.
1251
- 1252 **GrB_NULL_POINTER** indices or values pointer is `NULL`.
- 1253 **GrB_INDEX_OUT_OF_BOUNDS** A value in indices is outside the allowed range for **w**.
- 1254 **GrB_DOMAIN_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are not
1255 all the same, or the domains of values and **w** are incompatible with
1256 each other or D_{dup} .
- 1257 **GrB_OUTPUT_NOT_EMPTY** Output vector **w** already contains valid tuples (elements). In other
1258 words, `GrB_Vector_nvals(C)` returns a positive value.

1259 **Description**

1260 An internal vector $\tilde{\mathbf{w}} = \langle D_{dup}, \mathbf{size}(\mathbf{w}), \emptyset \rangle$ is created, which only differs from \mathbf{w} in its domain.
1261 Each tuple $\{\text{indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the output in the form of

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

1262 If multiple values for the same location are present in the input arrays, the `dup` binary operand is
1263 used to reduce them before assignment into $\tilde{\mathbf{w}}$ as follows:

1264

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

1265 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{w}}$ is copied into \mathbf{w} via typecasting its
1266 values to $\mathbf{D}(\mathbf{w})$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

1267 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1268 It is an error to call this function on an output object with existing elements. In other words,
1269 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1270 After `GrB.Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices` or
1271 `values`.

1272 **4.2.2.8 Vector_setElement: Set a single element in a vector**

1273 Set one element of a vector to a given value.

1274 **C Syntax**

```
1275     GrB_Info GrB_Vector_setElement(GrB_Vector  w,  
1276                                 <type>     val,  
1277                                 GrB_Index   index);
```

1278 **Parameters**

1279 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1280 `val` (IN) Scalar value to assign. The type must be compatible with the domain of `w`.

1281 `index` (IN) The location of the element to be assigned.

1282 Return Values

1283 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1284 blocking mode, this indicates that the compatibility tests on in-
1285 dex/dimensions and domains for the input arguments passed suc-
1286 cessfully. Either way, the output vector `w` is ready to be used in
1287 the next method of the sequence.

1288 GrB_PANIC Unknown internal error.

1289 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1290 GraphBLAS objects (input or output) is in an invalid state caused
1291 by a previous execution error. Call `GrB_error()` to access any error
1292 messages generated by the implementation.

1293 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1294 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `w`, has not been initialized by a call to
1295 `Vector_new` or `Vector_dup`.

1296 GrB_INVALID_INDEX `index` specifies a location that is outside the dimensions of `w`.

1297 GrB_DOMAIN_MISMATCH The domains of `w` and `val` are incompatible.

1298 Description

1299 First, the scalar and output vector are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ must be
1300 compatible with $\mathbf{D}(w)$. Two domains are compatible with each other if values from one domain can
1301 be cast to values in the other domain as per the rules of the C language. In particular, domains from
1302 Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible
1303 with itself. If any compatibility rule above is violated, execution of `GrB_Vector_setElement` ends and
1304 the domain mismatch error listed above is returned.

1305 Then, the `index` parameter is checked for a valid value where the following condition must hold:

$$1306 \quad 0 \leq \text{index} < \text{size}(w)$$

1307 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the invalid index error
1308 listed above is returned.

1309 We are now ready to carry out the assignment `val`; that is:

$$1310 \quad w(\text{index}) = \text{val}$$

1311 If a value existed at this location in `w`, it will be overwritten; otherwise, and new value is stored in
1312 `w`.

1313 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1314 of `w` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1315 return value `GrB_SUCCESS` and the new content of vector `w` is as defined above but may not be
1316 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1317 **4.2.2.9 Vector_removeElement: Remove an element from a vector**

1318 Remove (annihilate) one stored element from a vector.

1319 **C Syntax**

```
1320         GrB_Info GrB_Vector_removeElement(GrB_Vector  w,  
1321                                         GrB_Index   index);
```

1322 **Parameters**

1323 w (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1324 index (IN) The location of the element to be removed.

1325 **Return Values**

1326 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1327 blocking mode, this indicates that the compatibility tests on in-
1328 dex/dimensions and domains for the input arguments passed suc-
1329 cessfully. Either way, the output vector w is ready to be used in
1330 the next method of the sequence.

1331 GrB_PANIC Unknown internal error.

1332 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1333 GraphBLAS objects (input or output) is in an invalid state caused
1334 by a previous execution error. Call GrB_error() to access any error
1335 messages generated by the implementation.

1336 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1337 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, w, has not been initialized by a call to
1338 Vector_new or Vector_dup.

1339 GrB_INVALID_INDEX index specifies a location that is outside the dimensions of w.

1340 **Description**

1341 First, the index parameter is checked for a valid value where the following condition must hold:

$$1342 \quad 0 \leq \text{index} < \text{size}(w)$$

1343 If this condition is violated, execution of GrB_Vector_removeElement ends and the invalid index error
1344 listed above is returned.

1345 We are now ready to carry out the removal of a value that may be stored at the location specified
1346 by `index`. If a value does not exist at the specified location in `w`, no error is reported and the
1347 operation has no effect on the state of `w`. In either case, the following will be true on return from
1348 the method: `index` \notin `ind(w)`.

1349 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1350 of `w` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1351 return value `GrB_SUCCESS` and the new content of vector `w` is as defined above but may not be
1352 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1353 4.2.2.10 `Vector_extractElement`: Extract a single element from a vector.

1354 Extract one element of a vector into a scalar.

1355 C Syntax

```
1356         GrB_Info GrB_Vector_extractElement(<type>          *val,  
1357                                         const GrB_Vector  u,  
1358                                         GrB_Index         index);
```

1359 Parameters

1360 `val` (INOUT) Pointer to a scalar of type that is compatible with the domain of vector
1361 `w`. On successful return, this scalar holds the result of the operation. Any previous
1362 value in `val` is overwritten.

1363 `u` (IN) The GraphBLAS vector from which an element is extracted.

1364 `index` (IN) The location in `u` to extract.

1365 Return Values

1366 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
1367 cessfully. This indicates that the compatibility tests on dimensions
1368 and domains for the input arguments passed successfully, and the
1369 output scalar, `val`, has been computed and is ready to be used in
1370 the next method of the sequence.

1371 `GrB_PANIC` Unknown internal error.

1372 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1373 GraphBLAS objects (input or output) is in an invalid state caused
1374 by a previous execution error. Call `GrB_error()` to access any error
1375 messages generated by the implementation.

1376 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1377 GrB_UNINITIALIZED_OBJECT The GraphBLAS vector, `u`, has not been initialized by a call to
1378 `Vector_new` or `Vector_dup`.

1379 GrB_NULL_POINTER `val` pointer is NULL.

1380 GrB_NO_VALUE There is no stored value at specified location.

1381 GrB_INVALID_INDEX `index` specifies a location that is outside the dimensions of `w`.

1382 GrB_DOMAIN_MISMATCH The domains of the vector or scalar are incompatible.

1383 Description

1384 First, the scalar and input vector are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ must be
1385 compatible with $\mathbf{D}(u)$. Two domains are compatible with each other if values from one domain can
1386 be cast to values in the other domain as per the rules of the C language. In particular, domains from
1387 Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible
1388 with itself. If any compatibility rule above is violated, execution of `GrB_Vector_extractElement` ends
1389 and the domain mismatch error listed above is returned.

1390 Then, the `index` parameter is checked for a valid value where the following condition must hold:

$$1391 \quad 0 \leq \text{index} < \text{size}(u)$$

1392 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the invalid index error
1393 listed above is returned.

1394 We are now ready to carry out the extract into the output argument, `val`; that is:

$$1395 \quad \text{val} = u(\text{index})$$

1396 where the following condition must be true:

$$1397 \quad \text{index} \in \mathbf{ind}(u)$$

1398 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the "no value" error
1399 listed above is returned.

1400 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1401 `GrB_SUCCESS`, the new contents of `val` are as defined above.

1402 4.2.2.11 `Vector_extractTuples`: Extract tuples from a vector

1403 Extract the contents of a GraphBLAS vector into non-opaque data structures.

1404 **C Syntax**

```
1405     GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,  
1406                                     <type>         *values,  
1407                                     GrB_Index      *n,  
1408                                     const GrB_Vector v);  
1409
```

1410 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored
1411 values' indices.

1412 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
1413 the stored values whose type is compatible with **D(v)**.

1414 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the
1415 **values** and **indices** arrays can hold. Upon return, it will contain the number of
1416 values written to the arrays.

1417 **v** (IN) An existing GraphBLAS vector.

1418 **Return Values**

1419 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
1420 cessfully. This indicates that the compatibility tests on the input
1421 argument passed successfully, and the output arrays, **indices** and
1422 **values**, have been computed.

1423 **GrB_PANIC** Unknown internal error.

1424 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
1425 GraphBLAS objects (input or output) is in an invalid state caused
1426 by a previous execution error. Call **GrB_error()** to access any error
1427 messages generated by the implementation.

1428 **GrB_OUT_OF_MEMORY** Not enough memory available for operation.

1429 **GrB_INSUFFICIENT_SPACE** Not enough space in **indices** and **values** (as indicated by the **n** pa-
1430 rameter) to hold all of the tuples that will be extracted.

1431 **GrB_UNINITIALIZED_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to
1432 **Vector_new** or **Vector_dup**.

1433 **GrB_NULL_POINTER** **indices**, **values**, or **n** pointer is **NULL**.

1434 **GrB_DOMAIN_MISMATCH** The domains of the **v** vector or **values** array are incompatible with
1435 one another.

1436 **Description**

1437 This method will extract all the tuples from the GraphBLAS vector `v`. The values associated
1438 with those tuples are placed in the `values` array and the indices are placed in the `indices` array.
1439 Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least
1440 `GrB_Vector_nvals(v)` elements before calling this function.

1441 Upon return of this function, `n` will be set to the number of values (and indices) copied. Also, the
1442 entries of `indices` are unique, but not necessarily sorted. Each tuple (i, v_i) in `v` is unzipped and
1443 copied into a distinct k th location in output vectors:

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

1444 where $0 \leq k < \text{GrB_Vector_nvals}(v)$. No gaps in output vectors are allowed; that is, if `indices[k]` and
1445 `values[k]` exist upon return, so does `indices[j]` and `values[j]` for all j such that $0 \leq j < k$.

1446 Note that if the value in `n` on input is less than the number of values contained in the vector `v`,
1447 then a `GrB_INSUFFICIENT_SPACE` error is returned because it is undefined which subset of values
1448 would be extracted otherwise.

1449 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1450 `GrB_SUCCESS`, the new contents of the arrays `indices` and `values` are as defined above.

1451 **4.2.3 Matrix Methods**

1452 **4.2.3.1 Matrix_new: Create new matrix**

1453 Creates a new matrix with specified domain and dimensions.

1454 **C Syntax**

```
1455     GrB_Info GrB_Matrix_new(GrB_Matrix *A,  
1456                           GrB_Type    d,  
1457                           GrB_Index   nrows,  
1458                           GrB_Index   ncols);
```

1459 **Parameters**

1460 `A` (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1461 matrix.

1462 `d` (IN) The type corresponding to the domain of the matrix being created. Can be
1463 one of the predefined GraphBLAS types in Table 2.2, or an existing user-defined
1464 GraphBLAS type.

1465 `nrows` (IN) The number of rows of the matrix being created.

1466 ncols (IN) The number of columns of the matrix being created.

1467 **Return Values**

1468 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1469 blocking mode, this indicates that the API checks for the input ar-
1470 guments passed successfully. Either way, output matrix **A** is ready
1471 to be used in the next method of the sequence.

1472 GrB_PANIC Unknown internal error.

1473 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1474 GraphBLAS objects (input or output) is in an invalid state caused
1475 by a previous execution error. Call `GrB_error()` to access any error
1476 messages generated by the implementation.

1477 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1478 GrB_UNINITIALIZED_OBJECT The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`
1479 (needed for user-defined types).

1480 GrB_NULL_POINTER The **A** pointer is NULL.

1481 GrB_INVALID_VALUE `nrows` or `ncols` is zero.

1482 **Description**

1483 Creates a new matrix **A** of domain **D**(**d**), size `nrows` × `ncols`, and empty **L**(**A**). The method returns
1484 a handle to the new matrix in **A**.

1485 It is not an error to call this method more than once on the same variable; however, the handle to
1486 the previously created object will be overwritten.

1487 **4.2.3.2 Matrix_dup: Create a copy of a GraphBLAS matrix**

1488 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

1489 **C Syntax**

```
1490           GrB_Info GrB_Matrix_dup(GrB_Matrix        *C,  
1491                                    const GrB_Matrix  A);
```

1492 **Parameters**

1493 **C** (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1494 matrix.

1495 A (IN) The GraphBLAS matrix to be duplicated.

1496 Return Values

1497 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1498 blocking mode, this indicates that the API checks for the input
1499 arguments passed successfully. Either way, output matrix C is ready
1500 to be used in the next method of the sequence.

1501 GrB_PANIC Unknown internal error.

1502 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1503 GraphBLAS objects (input or output) is in an invalid state caused
1504 by a previous execution error. Call GrB_error() to access any error
1505 messages generated by the implementation.

1506 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1507 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
1508 Matrix_new or Matrix_dup.

1509 GrB_NULL_POINTER The C pointer is NULL.

1510 Description

1511 Creates a new matrix **C** of domain $\mathbf{D}(A)$, size $\mathbf{nrows}(A) \times \mathbf{ncols}(A)$, and contents $\mathbf{L}(A)$. It returns
1512 a handle to it in C.

1513 It is not an error to call this method more than once on the same variable; however, the handle to
1514 the previously created object will be overwritten.

1515 4.2.3.3 Matrix_resize: Resize a matrix

1516 Changes the dimensions of an existing matrix.

1517 C Syntax

```
1518 GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
1519                             GrB_Index nrows,  
1520                             GrB_Index ncols);
```

1521 Parameters

1522 C (INOUT) An existing Matrix object that is being resized.

1523 `nrows` (IN) The new number of rows of the matrix. It can be smaller or larger than the
1524 current number of rows.

1525 `ncols` (IN) The new number of columns of the matrix. It can be smaller or larger than
1526 the current number of columns.

1527 **Return Values**

1528 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
1529 blocking mode, this indicates that the API checks for the input
1530 arguments passed successfully. Either way, output matrix `C` is ready
1531 to be used in the next method of the sequence.

1532 `GrB_PANIC` Unknown internal error.

1533 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1534 GraphBLAS objects (input or output) is in an invalid state caused
1535 by a previous execution error. Call `GrB_error()` to access any error
1536 messages generated by the implementation.

1537 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1538 `GrB_NULL_POINTER` The `C` pointer is `NULL`.

1539 `GrB_INVALID_VALUE` `nrows` or `ncols` is zero.

1540 **Description**

1541 Changes the number of rows and columns of `C` to `nrows` and `ncols`, respectively. The domain $\mathbf{D}(C)$
1542 of matrix `C` remains the same. The contents $\mathbf{L}(C)$ are modified as described below.

1543 Let $C = \langle \mathbf{D}(C), M, N, \mathbf{L}(C) \rangle$ when the method is called. When the method returns `C` is modified
1544 to $C = \langle \mathbf{D}(C), \text{nrows}, \text{ncols}, \mathbf{L}'(C) \rangle$ where $\mathbf{L}'(C) = \{(i, j, C_{ij}) : (i, j, C_{ij}) \in \mathbf{L}(C) \wedge (i < \text{nrows}) \wedge (j < \text{ncols})\}$. That is, all elements of `C` with row index greater than or equal to `nrows` or column index
1545 greater than or equal to `ncols` are dropped.
1546

1547 **4.2.3.4 Matrix_clear: Clear a matrix**

1548 Removes all elements (tuples) from a matrix.

1549 **C Syntax**

1550 `GrB_Info GrB_Matrix_clear(GrB_Matrix A);`

1551 **Parameters**

1552 A (IN) An existing GraphBLAS matrix to clear.

1553 **Return Values**

1554 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1555 blocking mode, this indicates that the API checks for the input ar-
1556 guments passed successfully. Either way, output matrix A is ready
1557 to be used in the next method of the sequence.

1558 GrB_PANIC Unknown internal error.

1559 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1560 GraphBLAS objects (input or output) is in an invalid state caused
1561 by a previous execution error. Call GrB_error() to access any error
1562 messages generated by the implementation.

1563 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1564 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, *A, has not been initialized by a call to
1565 Matrix_new or Matrix_dup.

1566 **Description**

1567 Removes all elements (tuples) from an existing matrix. After the call to GrB_Matrix_clear(A),
1568 $L(\mathbf{A}) = \emptyset$. The dimensions of the matrix do not change.

1569 **4.2.3.5 Matrix_nrows: Number of rows in a matrix**

1570 Retrieve the number of rows in a matrix.

1571 **C Syntax**

```
1572           GrB_Info GrB_Matrix_nrows(GrB_Index            *nrows,  
1573                                       const GrB_Matrix  A);
```

1574 **Parameters**

1575 nrows (OUT) On successful return, contains the number of rows in the matrix.

1576 A (IN) An existing GraphBLAS matrix being queried.

1577 **Return Values**

1578 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1579 cessfully and the value of `nrows` has been set.

1580 GrB_PANIC Unknown internal error.

1581 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1582 GraphBLAS objects (input or output) is in an invalid state caused
1583 by a previous execution error. Call `GrB_error()` to access any error
1584 messages generated by the implementation.

1585 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to
1586 `Matrix_new` or `Matrix_dup`.

1587 GrB_NULL_POINTER `nrows` pointer is NULL.

1588 **Description**

1589 Return `nrows(A)` in `nrows` (the number of rows).

1590 **4.2.3.6 Matrix_ncols: Number of columns in a matrix**

1591 Retrieve the number of columns in a matrix.

1592 **C Syntax**

```
1593                   GrB_Info GrB_Matrix_ncols(GrB_Index           *ncols,  
1594                                            const GrB_Matrix A);
```

1595 **Parameters**

1596 `ncols` (OUT) On successful return, contains the number of columns in the matrix.

1597 `A` (IN) An existing GraphBLAS matrix being queried.

1598 **Return Values**

1599 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1600 cessfully and the value of `ncols` has been set.

1601 GrB_PANIC Unknown internal error.

1602 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1603 GraphBLAS objects (input or output) is in an invalid state caused
1604 by a previous execution error. Call GrB_error() to access any error
1605 messages generated by the implementation.

1606 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
1607 Matrix_new or Matrix_dup.

1608 GrB_NULL_POINTER ncols pointer is NULL.

1609 **Description**

1610 Return `ncols(A)` in `ncols` (the number of columns).

1611 **4.2.3.7 Matrix_nvals: Number of stored elements in a matrix**

1612 Retrieve the number of stored elements (tuples) in a matrix.

1613 **C Syntax**

```
1614           GrB_Info GrB_Matrix_nvals(GrB_Index            *nvals,
1615                                                            const GrB_Matrix A);
```

1616 **Parameters**

1617 nvals (OUT) On successful return, contains the number of stored elements (tuples) in
1618 the matrix.

1619 A (IN) An existing GraphBLAS matrix being queried.

1620 **Return Values**

1621 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1622 cessfully and the value of `nvals` has been set.

1623 GrB_PANIC Unknown internal error.

1624 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1625 GraphBLAS objects (input or output) is in an invalid state caused
1626 by a previous execution error. Call GrB_error() to access any error
1627 messages generated by the implementation.

1628 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1629 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
1630 Matrix_new or Matrix_dup.

1631 GrB_NULL_POINTER The nvals pointer is NULL.

1632 Description

1633 Return `nvals(A)` in `nvals`. This is the number of tuples stored in matrix A, which is the size of
1634 $\mathbf{L}(\mathbf{A})$ (see Section 3.5).

1635 4.2.3.8 Matrix_build: Store elements from tuples into a matrix

1636 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);
```

1637 Parameters

1638 C (INOUT) An existing Matrix object to store the result.

1639 row_indices (IN) Pointer to an array of row indices.

1640 col_indices (IN) Pointer to an array of column indices.

1641 values (IN) Pointer to an array of scalars of a type that is compatible with the domain of
1642 matrix, C.

1643 n (IN) The number of entries contained in each array (the same for row_indices,
1644 col_indices, and values).

1645 dup (IN) An associative and commutative binary function to apply when duplicate
1646 values for the same location are present in the input arrays. All three domains of
1647 dup must be the same; hence $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$.

1648 Return Values

1649 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1650 blocking mode, this indicates that the API checks for the input
1651 arguments passed successfully. Either way, output matrix C is ready
1652 to be used in the next method of the sequence.

1653 GrB_PANIC Unknown internal error.

1654 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1655 GraphBLAS objects (input or output) is in an invalid state caused
1656 by a previous execution error. Call `GrB_error()` to access any error
1657 messages generated by the implementation.

1658 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1659 GrB_UNINITIALIZED_OBJECT Either `C` has not been initialized by a call to `GrB_Matrix_new` or
1660 by `GrB_Matrix_dup`, or `dup` has not been initialized by a call to `GrB_BinaryOp_new`.
1661

1662 GrB_NULL_POINTER `row_indices`, `col_indices` or `values` pointer is `NULL`.

1663 GrB_INDEX_OUT_OF_BOUNDS A value in `row_indices` or `col_indices` is outside the allowed range for
1664 `C`.

1665 GrB_DOMAIN_MISMATCH Either the domains of the GraphBLAS binary operator `dup` are not
1666 all the same, or the domains of `values` and `C` are incompatible with
1667 each other or D_{dup} .

1668 GrB_OUTPUT_NOT_EMPTY Output matrix `C` already contains valid tuples (elements). In other
1669 words, `GrB_Matrix_nvals(C)` returns a positive value.

1670 Description

1671 An internal matrix $\tilde{\mathbf{C}} = \langle D_{dup}, \mathbf{nrows}(C), \mathbf{ncols}(C), \emptyset \rangle$ is created, which only differs from `C` in its
1672 domain.

1673 Each tuple $\{\text{row_indices}[k], \text{col_indices}[k], \text{values}[k]\}$, where $0 \leq k < n$, is a contribution to the output
1674 in the form of

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

1675 If multiple values for the same location are present in the input arrays, the `dup` binary operand is
1676 used to reduce them before assignment into $\tilde{\mathbf{C}}$ as follows:

$$1677 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row_indices}[k]=i \wedge \text{col_indices}[k]=j} (D_{dup}) \text{values}[k],$$

1678 where \oplus is the `dup` binary operator. Finally, the resulting $\tilde{\mathbf{C}}$ is copied into `C` via typecasting its
1679 values to $\mathbf{D}(C)$ if necessary. If \oplus is not associative or not commutative, the result is undefined.

1680 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

1681 It is an error to call this function on an output object with existing elements. In other words,
1682 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

1683 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,
1684 `col_indices`, or `values`.

1685 **4.2.3.9 Matrix_setElement: Set a single element in matrix**

1686 Set one element of a matrix to a given value.

1687 **C Syntax**

```
1688         GrB_Info GrB_Matrix_setElement(GrB_Matrix  C,  
1689                                     <type>      val,  
1690                                     GrB_Index   row_index,  
1691                                     GrB_Index   col_index);
```

1692 **Parameters**

1693 C (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.

1694 val (IN) Scalar value to assign. The type must be compatible with the domain of C.

1695 row_index (IN) Row index of element to be assigned

1696 col_index (IN) Column index of element to be assigned

1697 **Return Values**

1698 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
1699 blocking mode, this indicates that the compatibility tests on in-
1700 dex/dimensions and domains for the input arguments passed suc-
1701 cessfully. Either way, the output matrix C is ready to be used in
1702 the next method of the sequence.

1703 GrB_PANIC Unknown internal error.

1704 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1705 GraphBLAS objects (input or output) is in an invalid state caused
1706 by a previous execution error. Call `GrB_error()` to access any error
1707 messages generated by the implementation.

1708 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1709 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to
1710 `Matrix_new` or `Matrix_dup`.

1711 GrB_INVALID_INDEX `row_index` or `col_index` is outside the allowable range (i.e., not less
1712 than `nrows(C)` or `ncols(C)`, respectively).

1713 GrB_DOMAIN_MISMATCH The domains of C and `val` are incompatible.

1714 **Description**

1715 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** must be
1716 compatible with **D(C)**. Two domains are compatible with each other if values from one domain can
1717 be cast to values in the other domain as per the rules of the C language. In particular, domains from
1718 Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible
1719 with itself. If any compatibility rule above is violated, execution of `GrB_Matrix_extractElement` ends
1720 and the domain mismatch error listed above is returned.

1721 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 1722 \quad & 0 \leq \text{row_index} < \mathbf{nrows}(C), \\ & 0 \leq \text{col_index} < \mathbf{ncols}(C) \end{aligned}$$

1723 If either of these conditions is violated, execution of `GrB_Matrix_extractElement` ends and the invalid
1724 index error listed above is returned.

1725 We are now ready to carry out the assignment of `val`; that is,

$$1726 \quad C(\text{row_index}, \text{col_index}) = \text{val}$$

1727 If a value existed at this location in `C`, it will be overwritten; otherwise, new value is stored in
1728 `C`.

1729 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents
1730 of `C` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with
1731 return value `GrB_SUCCESS` and the new content of vector `C` is as defined above but may not be
1732 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

1733 **4.2.3.10 Matrix_removeElement: Remove an element from a matrix**

1734 Remove (annihilate) one stored element from a matrix.

1735 **C Syntax**

```
1736     GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,  
1737                                     GrB_Index   row_index,  
1738                                     GrB_Index   col_index);
```

1739 **Parameters**

1740 `C` (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

1741 `row_index` (IN) Row index of element to be removed

1742 `col_index` (IN) Column index of element to be removed

1774 **C Syntax**

```
1775         GrB_Info GrB_Matrix_extractElement(<type>          *val,  
1776                                         const GrB_Matrix A,  
1777                                         GrB_Index      row_index,  
1778                                         GrB_Index      col_index);  
1779
```

1780 **Parameters**

1781 val (OUT) Pointer to a scalar of type that is compatible with the domain of matrix A.
1782 On successful return, this scalar holds the result of the operation. Any previous
1783 value in val is overwritten.

1784 A (IN) The GraphBLAS matrix from which an element is extracted.

1785 row_index (IN) The row index of location in A to extract.

1786 col_index (IN) The column index of location in A to extract.

1787 **Return Values**

1788 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
1789 cessfully. This indicates that the compatibility tests on dimensions
1790 and domains for the input arguments passed successfully, and the
1791 output scalar, val, has been computed and is ready to be used in
1792 the next method of the sequence.

1793 GrB_PANIC Unknown internal error.

1794 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
1795 GraphBLAS objects (input or output) is in an invalid state caused
1796 by a previous execution error. Call GrB_error() to access any error
1797 messages generated by the implementation.

1798 GrB_OUT_OF_MEMORY Not enough memory available for operation.

1799 GrB_UNINITIALIZED_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to
1800 Matrix_new or Matrix_dup.

1801 GrB_NULL_POINTER val pointer is NULL.

1802 GrB_NO_VALUE There is no stored value at specified location.

1803 GrB_INVALID_INDEX row_index or col_index is outside the allowable range (i.e. less than
1804 zero or greater than or equal to nrows(A) or ncols(A), respec-
1805 tively).

1806 GrB_DOMAIN_MISMATCH The domains of the matrix and scalar are incompatible.

1807 **Description**

1808 First, the scalar and input matrix are tested for domain compatibility as follows: $\mathbf{D}(\text{val})$ must be
1809 compatible with $\mathbf{D}(A)$. Two domains are compatible with each other if values from one domain can
1810 be cast to values in the other domain as per the rules of the C language. In particular, domains from
1811 Table 2.2 are all compatible with each other. A domain from a user-defined type is only compatible
1812 with itself. If any compatibility rule above is violated, execution of `GrB_Matrix_extractElement` ends
1813 and the domain mismatch error listed above is returned.

1814 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 1815 \quad & 0 \leq \text{row_index} < \mathbf{nrows}(A), \\ & 0 \leq \text{col_index} < \mathbf{ncols}(A) \end{aligned}$$

1816 If either of these conditions is violated, execution of `GrB_Matrix_extractElement` ends and the invalid
1817 index error listed above is returned.

1818 We are now ready to carry out the extract into the output argument, `val`; that is,

$$1819 \quad \text{val} = A(\text{row_index}, \text{col_index})$$

1820 where the following condition must be true:

$$1821 \quad (\text{row_index}, \text{col_index}) \in \mathbf{ind}(A)$$

1822 If this condition is violated, execution of `GrB_Matrix_extractElement` ends and the "no value" error
1823 listed above is returned.

1824 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1825 `GrB_SUCCESS`, the new contents of `val` are as defined above.

1826 **4.2.3.12 Matrix_extractTuples: Extract tuples from a matrix**

1827 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

1828 **C Syntax**

```
1829     GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
1830                                     GrB_Index      *col_indices,
1831                                     <type>         *values,
1832                                     GrB_Index      *n,
1833                                     const GrB_Matrix A);
```

1834 **Parameters**

1835 `row_indices` (OUT) Pointer to an array of row indices that is large enough to hold all of the
1836 row indices.

1837 `col_indices` (OUT) Pointer to an array of column indices that is large enough to hold all of the
1838 column indices.

1839 `values` (OUT) Pointer to an array of scalars of a type that is large enough to hold all of
1840 the stored values whose type is compatible with $\mathbf{D}(\mathbf{A})$.

1841 `n` (INOUT) Pointer to a value indicating (in input) the number of elements the `values`,
1842 `row_indices`, and `col_indices` arrays can hold. Upon return, it will contain the number
1843 of values written to the arrays.

1844 `A` (IN) An existing GraphBLAS matrix.

1845 Return Values

1846 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-
1847 cessfully. This indicates that the compatibility tests on the input
1848 argument passed successfully, and the output arrays, indices and
1849 values, have been computed.

1850 `GrB_PANIC` Unknown internal error.

1851 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
1852 GraphBLAS objects (input or output) is in an invalid state caused
1853 by a previous execution error. Call `GrB_error()` to access any error
1854 messages generated by the implementation.

1855 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1856 `GrB_INSUFFICIENT_SPACE` Not enough space in `row_indices`, `col_indices`, and `values` (as indicated
1857 by the `n` parameter) to hold all of the tuples that will be extracted.

1858 `GrB_UNINITIALIZED_OBJECT` The GraphBLAS matrix, `A`, has not been initialized by a call to
1859 `Matrix_new` or `Matrix_dup`.

1860 `GrB_NULL_POINTER` `row_indices`, `col_indices`, `values` or `n` pointer is `NULL`.

1861 `GrB_DOMAIN_MISMATCH` The domains of the `A` matrix and `values` array are incompatible
1862 with one another.

1863 Description

1864 This method will extract all the tuples from the GraphBLAS matrix `A`. The values associated with
1865 those tuples are placed in the `values` array, the column indices are placed in the `col_indices` array,
1866 and the row indices are placed in the `row_indices` array. These output arrays are pre-allocated by
1867 the user before calling this function such that each output array has enough space to hold at least
1868 `GrB_Matrix_nvals(A)` elements.

1869 Upon return of this function, a pair of $\{\text{row_indices}[k], \text{col_indices}[k]\}$ are unique for every valid k ,
1870 but they are not required to be sorted in any particular order. Each tuple (i, j, A_{ij}) in A is unzipped
1871 and copied into a distinct k th location in output vectors:

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

1872 where $0 \leq k < \text{GrB_Matrix_nvals}(v)$. No gaps in output vectors are allowed; that is, if $\text{row_indices}[k]$,
1873 $\text{col_indices}[k]$ and $\text{values}[k]$ exist upon return, so does $\text{row_indices}[j]$, $\text{col_indices}[j]$ and $\text{values}[j]$ for all
1874 j such that $0 \leq j < k$.

1875 Note that if the value in n on input is less than the number of values contained in the matrix A ,
1876 then a `GrB_INSUFFICIENT_SPACE` error is returned since it is undefined which subset of values
1877 would be extracted.

1878 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value
1879 `GrB_SUCCESS`, the new contents of the arrays `row_indices`, `col_indices` and `values` are as defined
1880 above.

1881 4.2.4 Descriptor Methods

1882 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-
1883 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

1884 4.2.4.1 Descriptor_new: Create new descriptor

1885 Creates a new (empty or default) descriptor.

1886 C Syntax

```
1887 GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);
```

1888 Parameters

1889 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS
1890 descriptor.

1891 Return Value

1892 GrB_SUCCESS The method completed successfully.

1893 GrB_PANIC unknown internal error.

1894 GrB_OUT_OF_MEMORY not enough memory available for operation.

1895 GrB_NULL_POINTER desc pointer is NULL.

1896 **Description**

1897 Creates a new descriptor object and returns a handle to it in `desc`. A newly created descriptor can
1898 be populated by calls to `Descriptor_set`.

1899 It is not an error to call this method more than once on the same variable; however, the handle to
1900 the previously created object will be overwritten.

1901 **4.2.4.2 Descriptor_set: Set content of descriptor**

1902 Sets the content for a field for an existing descriptor.

1903 **C Syntax**

```
1904         GrB_Info GrB_Descriptor_set(GrB_Descriptor      desc,  
1905                                   GrB_Desc_Field      field,  
1906                                   GrB_Desc_Value      val);
```

1907 **Parameters**

1908 `desc` (IN) An existing GraphBLAS descriptor to be modified.

1909 `field` (IN) The field being set.

1910 `val` (IN) New value for the field being set.

1911 **Return Values**

1912 `GrB_SUCCESS` operation completed successfully.

1913 `GrB_PANIC` unknown internal error.

1914 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1915 `GrB_UNINITIALIZED_OBJECT` the `desc` parameter has not been initialized by a call to `new`.

1916 `GrB_INVALID_VALUE` invalid value set on the field, or invalid field.

1917 **Description**

1918 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor
1919 to set the value associated with that field. Valid values for the field parameter include the following:

1920 `GrB_OUTP` refers to the output parameter (result) of the operation.

1921 GrB_MASK refers to the mask parameter of the operation.
1922 GrB_INP0 refers to the first input parameters of the operation (matrices and vectors).
1923 GrB_INP1 refers to the second input parameters of the operation (matrices and vectors).

1924 Valid values for the val parameter are:

1925 GrB_STRUCTURE Use only the structure of the stored values of the corresponding mask
1926 (GrB_MASK) parameter.
1927 GrB_COMP Use the complement of the corresponding mask (GrB_MASK) param-
1928 eter. When combined with GrB_STRUCTURE, the complement of the
1929 structure of the mask is used without evaluating the values stored.
1930 GrB_TRAN Use the transpose of the corresponding matrix parameter (valid for input
1931 matrix parameters only).
1932 GrB_REPLACE When assigning the masked values to the output matrix or vector, clear
1933 the matrix first (or clear the non-masked entries). The default behavior
1934 is to leave non-masked locations unchanged. Valid for the GrB_OUTP
1935 parameter only.

1936 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of GrB_MASK,
1937 multiple values can be set and all will apply (for example, both GrB_COMP and GrB_STRUCTURE).
1938 A value for a given field may be set multiple times but will have no additional effect. Fields that
1939 have no values set result in their default behavior, as defined in Section 3.7.

1940 4.2.5 free method

1941 Destroys a previously created GraphBLAS object and releases any resources associated with the
1942 object.

1943 C Syntax

```
1944 GrB_Info GrB_free(GrB_Object *obj);
```

1945 Parameters

1946 obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have
1947 been created by an explicit call to a GraphBLAS constructor. Can be any of the
1948 opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,
1949 binary op, unary op, or type. On successful completion of GrB_free, obj behaves
1950 as an uninitialized object.

1951 **Return Values**

1952 `GrB_SUCCESS` operation completed successfully

1953 `GrB_PANIC` unknown internal error. If this return value is encountered when
1954 in nonblocking mode, the error responsible for the panic condition
1955 could be from any method involved in the computation of the input
1956 object. The `GrB_error()` method should be called for additional
1957 information.

1958 **Description**

1959 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime
1960 system. A call to `GrB_free` frees those resources so they are available for use by other GraphBLAS
1961 objects.

1962 The parameter passed into `GrB_free` is a handle referencing a GraphBLAS opaque object of a
1963 data type from table 2.1. The object must have been created by an explicit call to a GraphBLAS
1964 constructor. The behavior of a program that calls `GrB_free` on a pre-defined object is implementation
1965 defined.

1966 After the `GrB_free` method returns, the object referenced by the input handle is destroyed and the
1967 handle has the value `GrB_INVALID_HANDLE`. The handle can be used in subsequent GraphBLAS
1968 methods but only after the handle has been reinitialized with a call the the appropriate `_new` or
1969 `_dup` method.

1970 Note that unlike other GraphBLAS methods, calling `GrB_free` with an object with an invalid handle
1971 is legal. The system may attempt to free resources that might be associated with that object, if
1972 possible, and return normally.

1973 When using `GrB_free` it is possible to create a dangling reference to an object. This would occur
1974 when a handle is assigned to a second variable of the same opaque type. This creates two handles
1975 that reference the same object. If `GrB_free` is called with one of the variables, the object is destroyed
1976 and the handle associated with the other variable no longer references a valid object. This is not an
1977 error condition that the implementation of the GraphBLAS API can be expected to catch, hence
1978 programmers must take care to prevent this situation from occurring.

1979 **4.3 GraphBLAS Operations**

1980 The GraphBLAS operations are defined in the GraphBLAS math specification and summarized in
1981 Table 4.1. In addition to methods that implement these fundamental GraphBLAS operations, we
1982 support a number of variants that have been found to be especially useful in algorithm development.
1983 A flowchart of the overall behavior of a GraphBLAS operation is shown in Figure 4.1.

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 controls which values resulting from the operation on the right-hand side are written into the output object (complement and structure flags are not shown). 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	$\mathbf{C}\langle\mathbf{M}, z\rangle$	$= \mathbf{C} \odot \mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, z\rangle$	$= \mathbf{w} \odot \mathbf{u} \oplus \mathbf{v}$
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	$\mathbf{C}\langle\mathbf{M}, z\rangle(i, j)$	$= \mathbf{C}(i, j) \odot \mathbf{A}$
	$\mathbf{w}\langle\mathbf{m}, z\rangle(i)$	$= \mathbf{w}(i) \odot \mathbf{u}$
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	$\mathbf{C}\langle\mathbf{M}, z\rangle$	$= \mathbf{C} \odot \mathbf{A}^T$
kroncker	$\mathbf{C}\langle\mathbf{M}, z\rangle$	$= \mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

1984 Domains and Casting

1985 A GraphBLAS operation is only valid when the domains of the GraphBLAS objects are mathemat-
1986 ically consistent. The C programming language defines implicit casts between built-in data types.
1987 For example, floats, doubles, and ints can be freely mixed according to the rules defined for implicit
1988 casts. It is the responsibility of the user to assure that these casts are appropriate for the algorithm
1989 in question. For example, a cast to int implies truncation of a floating point type. Depending on
1990 the operation, this truncation error could lead to erroneous results. Furthermore, casting a wider
1991 type onto a narrower type can lead to overflow errors. The GraphBLAS operations do not attempt
1992 to protect a user from these sorts of errors.

1993 When user-define types are involved, however, GraphBLAS requires strict equivalence between
1994 types and no casting is supported. If GraphBLAS detects these mismatches, it will return a
1995 domain mismatch error.

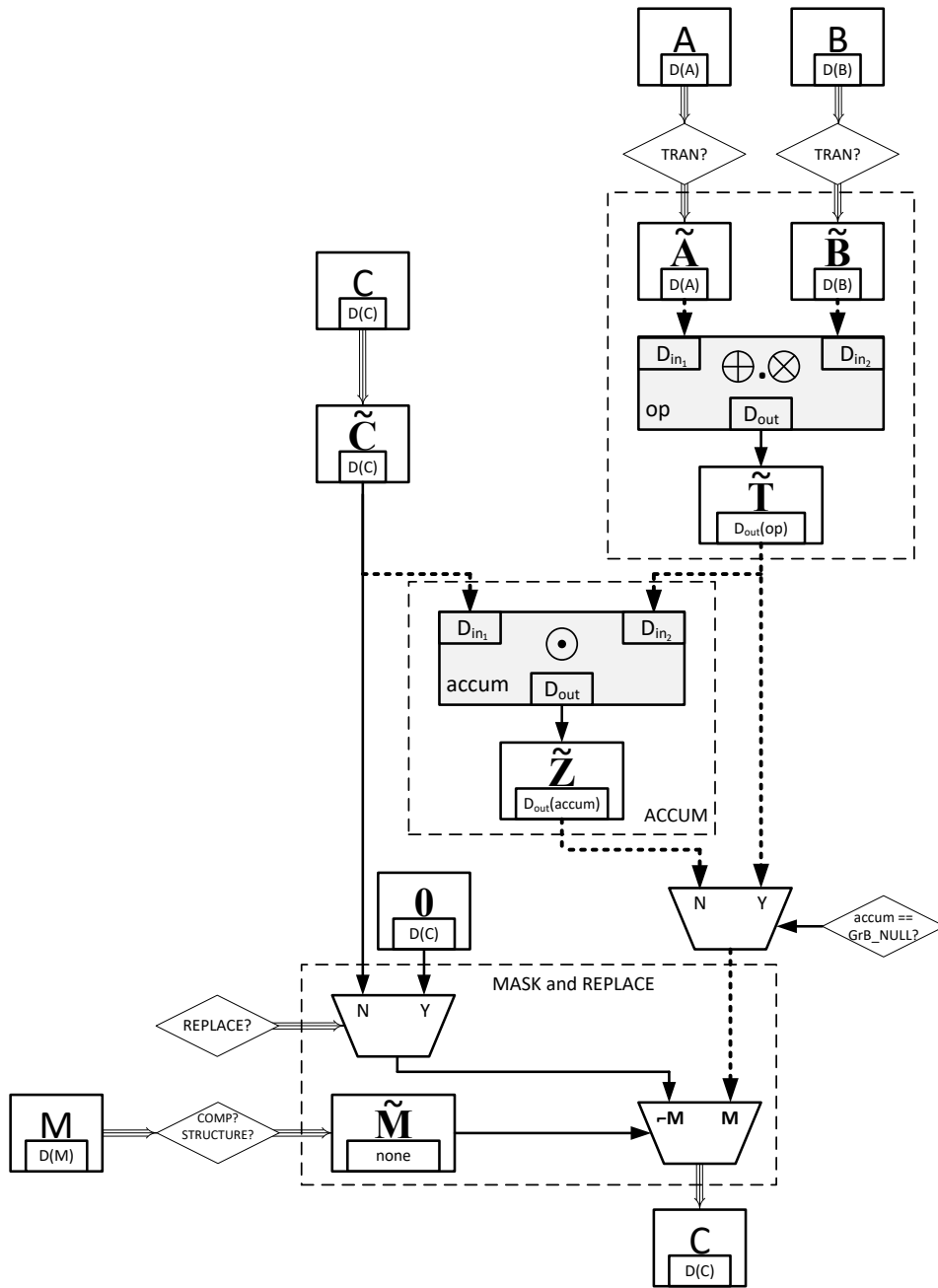


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows (\Rightarrow) denote where “as if copy” takes place (including both collections and descriptor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

1996 Dimensions and Transposes

1997 GraphBLAS operations also make assumptions about the numbers of dimensions and the sizes of
1998 vectors and matrices in an operation. An operation will test these sizes and report an error if they
1999 are not *shape compatible*. For example, when multiplying two matrices, $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, the number
2000 of rows of \mathbf{C} must equal the number of rows of \mathbf{A} , the number of columns of \mathbf{A} must match the
2001 number of rows of \mathbf{B} , and the number of columns of \mathbf{C} must match the number of columns of \mathbf{B} .
2002 This is the behavior expected given the mathematical definition of the operations.

2003 For most of the GraphBLAS operations involving matrices, an optional descriptor can modify the
2004 matrix associated with an input GraphBLAS matrix object. For example, if an input matrix is an
2005 argument to a GraphBLAS operation and the associated descriptor indicates the transpose option,
2006 then the operation occurs as if on the transposed matrix. In this case, the relationships between
2007 the sizes in each dimension shift in the mathematically expected way.

2008 Masks: Structure-only, Complement, and Replace

2009 When a GraphBLAS operation supports the use of an optional mask, that mask is specified through
2010 a GraphBLAS vector (for one-dimensional masks) or a GraphBLAS matrix (for two-dimensional
2011 masks). When a mask is used and the `GrB_STRUCTURE` descriptor value is not set, it is applied
2012 to the result from the operation wherever the stored values in the mask evaluate to true. If the
2013 `GrB_STRUCTURE` descriptor is set, the mask is applied to the result from the operation wherever the
2014 mask as a stored value (regardless of that value). Wherever the mask is applied, the result from
2015 the operation is either assigned to the provided output matrix/vector or, if a binary accumulation
2016 operation is provided, the result is accumulated into the corresponding elements of the provided
2017 output matrix/vector.

2018 Given a GraphBLAS vector $\mathbf{v} = \langle D, N, \{(i, v_i)\} \rangle$, a one-dimensional mask is derived for use in the
2019 operation as follows:

$$2020 \quad \mathbf{m} = \begin{cases} \langle N, \{\mathbf{ind}(\mathbf{v})\} \rangle, & \text{if } \text{GrB_STRUCTURE} \text{ is specified,} \\ \langle N, \{i : (\text{bool})v_i = \text{true}\} \rangle, & \text{otherwise} \end{cases}$$

2021 where $(\text{bool})v_i$ denotes casting the value v_i to a Boolean value (`true` or `false`). Likewise, given a
2022 GraphBLAS matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$, a two-dimensional mask is derived for use in the
2023 operation as follows:

$$2024 \quad \mathbf{M} = \begin{cases} \langle M, N, \{\mathbf{ind}(\mathbf{A})\} \rangle, & \text{if } \text{GrB_STRUCTURE} \text{ is specified,} \\ \langle M, N, \{(i, j) : (\text{bool})A_{ij} = \text{true}\} \rangle, & \text{otherwise} \end{cases}$$

2025 where $(\text{bool})A_{ij}$ denotes casting the value A_{ij} to a Boolean value. (`true` or `false`)

2026 In both the one- and two-dimensional cases, the mask may also have a subsequent complement
2027 operation applied (*Section 3.6*) as specified in the descriptor, before a final mask is generated for
2028 use in the operation.

2029 When the descriptor of an operation with a mask has specified that the `GrB_REPLACE` value is
2030 to be applied to the output (`GrB_OUTP`), then anywhere the mask is not true, the corresponding
2031 location in the output is cleared.

2032 Invalid and uninitialized objects

2033 Upon entering a GraphBLAS operation, the first step is a check that all objects are valid and ini-
2034 tialized. (Optional parameters can be set to `GrB_NULL`, which always counts as a valid object.) An
2035 invalid object is one that could not be computed due to a previous execution error. An uninitialized
2036 object is one that has not yet been created by a corresponding `new` or `dup` method. Appropriate
2037 error codes are returned if an object is not initialized (`GrB_UNINITIALIZED_OBJECT`) or invalid
2038 (`GrB_INVALID_OBJECT`).

2039 To support the detection of as many cases of uninitialized objects as possible, it is strongly recom-
2040 mended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at the
2041 point of their declaration, as shown in the following examples:

```
2042         GrB_Type          type = GrB_INVALID_HANDLE;  
2043         GrB_Semiring      semiring = GrB_INVALID_HANDLE;  
2044         GrB_Matrix       matrix = GrB_INVALID_HANDLE;
```

2045 Compliance

2046 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.
2047 That is, for each operation we give a recipe for producing its outcome. Any implementation that
2048 produces the same outcome, and follows the GraphBLAS execution model (Section 2.8) and error
2049 model (Section 2.9) is a conforming implementation.

2050 4.3.1 mxm: Matrix-matrix multiply

2051 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

2052 C Syntax

```
2053         GrB_Info GrB_mxm(GrB_Matrix          C,  
2054                         const GrB_Matrix   Mask,  
2055                         const GrB_BinaryOp  accum,  
2056                         const GrB_Semiring  op,  
2057                         const GrB_Matrix   A,  
2058                         const GrB_Matrix   B,  
2059                         const GrB_Descriptor desc);
```

2060 Parameters

2061 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
2062 that may be accumulated with the result of the matrix product. On output, the
2063 matrix holds the results of the operation.

2064 **Mask** (IN) An optional “write” mask that controls which results from this operation are
 2065 stored into the output matrix C. The mask dimensions must match those of the
 2066 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
 2067 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types
 2068 in Table 2.2. If the default mask is desired (i.e., a mask that is all `true` with the
 2069 dimensions of C), `GrB_NULL` should be specified.

2070 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 2071 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 2072 specified.

2073 **op** (IN) The semiring used in the matrix-matrix multiply.

2074 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 2075 multiplication.

2076 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
 2077 multiplication.

2078 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 2079 should be specified. Non-default field/value pairs are listed as follows:

2080

Param	Field	Value	Description
C	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <code>Mask</code> matrix. The stored values are not examined.
Mask	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <code>Mask</code> .
A	<code>GrB_INP0</code>	<code>GrB_TRAN</code>	Use transpose of A for the operation.
B	<code>GrB_INP1</code>	<code>GrB_TRAN</code>	Use transpose of B for the operation.

2081

2082 **Return Values**

2083 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 2084 blocking mode, this indicates that the compatibility tests on di-
 2085 mensions and domains for the input arguments passed successfully.
 2086 Either way, output matrix C is ready to be used in the next method
 2087 of the sequence.

2088 **GrB_PANIC** Unknown internal error.

2089 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 2090 GraphBLAS objects (input or output) is in an invalid state caused
 2091 by a previous execution error. Call `GrB_error()` to access any error
 2092 messages generated by the implementation.

2093 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2094 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
2095 a call to `new` (or `Matrix_dup` for matrix parameters).

2096 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

2097 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
2098 corresponding domains of the semiring or accumulation operator,
2099 or the mask's domain is not compatible with `bool` (in the case where
2100 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

2101 **Description**

2102 GrB_mxm computes the matrix product $C = A \oplus . \otimes B$ or, if an optional binary accumulation operator
2103 (\odot) is provided, $C = C \odot (A \oplus . \otimes B)$ (where matrices A and B can be optionally transposed).
2104 Logically, this operation occurs in three steps:

2105 **Setup** The internal matrices and mask used in the computation are formed and their domains
2106 and dimensions are tested for compatibility.

2107 **Compute** The indicated computations are carried out.

2108 **Output** The result is written into the output matrix, possibly under control of a mask.

2109 Up to four argument matrices are used in the GrB_mxm operation:

- 2110 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 2111 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)
- 2112 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 2113 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

2114 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for
2115 domain compatibility as follows:

- 2116 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
2117 must be from one of the pre-defined types of Table 2.2.
- 2118 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 2119 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 2120 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 2121 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
2122 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
2123 of the accumulation operator.

2124 Two domains are compatible with each other if values from one domain can be cast to values in
 2125 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
 2126 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 2127 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch
 2128 error listed above is returned.

2129 From the argument matrices, the internal matrices and mask used in the computation are formed
 2130 (\leftarrow denotes copy):

- 2131 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 2132 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 2133 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 2134 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 2135 (b) If `Mask \neq GrB_NULL`,
 - 2136 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 2137 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 2138 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 2139 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 2140 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 2141 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 2142 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

2143 The internal matrices and masks are checked for dimension compatibility. The following conditions
 2144 must hold:

- 2145 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 2146 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 2147 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 2148 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.
- 2149 5. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.

2150 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch
 2151 error listed above is returned.

2152 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 2153 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2154 We are now ready to carry out the matrix multiplication and any additional associated operations.
 2155 We describe this in terms of two intermediate matrices:

- 2156 • $\tilde{\mathbf{T}}$: The matrix holding the product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.

2157 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

2158 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:$
2159 $, j)) \neq \emptyset\}$ is created. The value of each of its elements is computed by

$$2160 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

2161 where \oplus and \otimes are the additive and multiplicative operators of semiring op , respectively.

2162 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

2163 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

2164 • If accum is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$2165 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

2166 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
2167 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$2168 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$2169 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$2170 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

2171 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

2172 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
2173 using what is called a *standard matrix mask and replace*. This is carried out under control of the
2174 mask which acts as a “write mask”.

2175 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is set, then any values in \mathbf{C} on input to this operation are
2176 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$2177 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

2178 • If $\text{desc}[\text{GrB_OUTP}].\text{GrB_REPLACE}$ is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
2179 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
2180 mask are unchanged:

$$2181 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

2182 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
2183 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
2184 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
2185 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
2186 sequence.

2189 **4.3.2 vxm: Vector-matrix multiply**

2190 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

2191 **C Syntax**

```
2192     GrB_Info GrB_vxm(GrB_Vector      w,  
2193                   const GrB_Vector  mask,  
2194                   const GrB_BinaryOp accum,  
2195                   const GrB_Semiring op,  
2196                   const GrB_Vector  u,  
2197                   const GrB_Matrix  A,  
2198                   const GrB_Descriptor desc);
```

2199 **Parameters**

2200 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
2201 that may be accumulated with the result of the vector-matrix product. On output,
2202 this vector holds the results of the operation.

2203 **mask** (IN) An optional “write” mask that controls which results from this operation are
2204 stored into the output vector *w*. The mask dimensions must match those of the
2205 vector *w*. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
2206 of the mask vector must be of type `bool` or any of the predefined “built-in” types
2207 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
2208 dimensions of *w*), `GrB_NULL` should be specified.

2209 **accum** (IN) An optional binary operator used for accumulating entries into existing *w*
2210 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
2211 specified.

2212 **op** (IN) Semiring used in the vector-matrix multiply.

2213 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the
2214 multiplication.

2215 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
2216 multiplication.

2217 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
2218 should be specified. Non-default field/value pairs are listed as follows:
2219

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation.

2220

2221 Return Values

2222 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 2223 blocking mode, this indicates that the compatibility tests on di-
 2224 mensions and domains for the input arguments passed successfully.
 2225 Either way, output vector w is ready to be used in the next method
 2226 of the sequence.

2227 GrB_PANIC Unknown internal error.

2228 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 2229 GraphBLAS objects (input or output) is in an invalid state caused
 2230 by a previous execution error. Call GrB_error() to access any error
 2231 messages generated by the implementation.

2232 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2233 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 2234 a call to new (or dup for matrix or vector parameters).

2235 GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

2236 GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with
 2237 the corresponding domains of the semiring or accumulation opera-
 2238 tor, or the mask's domain is not compatible with bool (in the case
 2239 where desc[GrB_MASK].GrB_STRUCTURE is not set).

2240 Description

2241 GrB_vxm computes the vector-matrix product $w^T = u^T \oplus . \otimes A$, or, if an optional binary accumulation
 2242 operator (\odot) is provided, $w^T = w^T \odot (u^T \oplus . \otimes A)$ (where matrix A can be optionally transposed).
 2243 Logically, this operation occurs in three steps:

2244 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
 2245 domains/dimensions are tested for compatibility.

2246 **Compute** The indicated computations are carried out.

2247 **Output** The result is written into the output vector, possibly under control of a mask.

2248 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 2249 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 2250 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 2251 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 2252 4. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

2253 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
2254 tested for domain compatibility as follows:

- 2255 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
2256 must be from one of the pre-defined types of Table 2.2.
- 2257 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{op})$ of the semiring.
- 2258 3. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{op})$ of the semiring.
- 2259 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\mathbf{op})$ of the semiring.
- 2260 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
2261 of the accumulation operator and $\mathbf{D}_{out}(\mathbf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$
2262 of the accumulation operator.

2263 Two domains are compatible with each other if values from one domain can be cast to values in
2264 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
2265 all compatible with each other. A domain from a user-defined type is only compatible with itself.
2266 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch
2267 error listed above is returned.

2268 From the argument vectors and matrices, the internal matrices and mask used in the computation
2269 are formed (\leftarrow denotes copy):

- 2270 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 2271 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 2272 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 2273 (b) If `mask` \neq `GrB_NULL`,
 - 2274 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 2275 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 2276 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 2277 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

2278 4. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

2279 The internal matrices and masks are checked for shape compatibility. The following conditions
2280 must hold:

2281 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$.

2282 2. $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$.

2283 3. $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$.

2284 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch
2285 error listed above is returned.

2286 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
2287 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2288 We are now ready to carry out the vector-matrix multiplication and any additional associated
2289 operations. We describe this in terms of two intermediate vectors:

2290 • $\tilde{\mathbf{t}}$: The vector holding the product of vector $\tilde{\mathbf{u}}^T$ and matrix $\tilde{\mathbf{A}}$.

2291 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

2292 The intermediate vector $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{ncols}(\tilde{\mathbf{A}}), \{(j, t_j) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j)) \neq \emptyset\} \rangle$ is created.
2293 The value of each of its elements is computed by

$$2294 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

2295 where \oplus and \otimes are the additive and multiplicative operators of semiring `op`, respectively.

2296 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

2297 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

2298 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$2299 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

2300 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
2301 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$2302 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

2303

$$2304 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

2305

$$2306 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

2307 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

2308 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 2309 using what is called a *standard vector mask and replace*. This is carried out under control of the
 2310 mask which acts as a “write mask”.

- 2311 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 2312 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$2313 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 2314 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 2315 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 2316 mask are unchanged:

$$2317 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

2318 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 2319 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 2320 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 2321 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

2322 4.3.3 mxv: Matrix-vector multiply

2323 Multiplies a matrix by a vector on a semiring. The result is a vector.

2324 C Syntax

```
2325     GrB_Info GrB_mxv(GrB_Vector      w,
2326                    const GrB_Vector mask,
2327                    const GrB_BinaryOp accum,
2328                    const GrB_Semiring op,
2329                    const GrB_Matrix  A,
2330                    const GrB_Vector  u,
2331                    const GrB_Descriptor desc);
```

2332 Parameters

2333 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 2334 that may be accumulated with the result of the matrix-vector product. On output,
 2335 this vector holds the results of the operation.

2336 **mask** (IN) An optional “write” mask that controls which results from this operation are
 2337 stored into the output vector \mathbf{w} . The mask dimensions must match those of the
 2338 vector \mathbf{w} . If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
 2339 of the mask vector must be of type `bool` or any of the predefined “built-in” types
 2340 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
 2341 dimensions of \mathbf{w}), `GrB_NULL` should be specified.

2342 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
 2343 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 2344 specified.

2345 **op** (IN) Semiring used in the vector-matrix multiply.

2346 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
 2347 multiplication.

2348 **u** (IN) The GraphBLAS vector holding the values for the right-hand vector in the
 2349 multiplication.

2350 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 2351 should be specified. Non-default field/value pairs are listed as follows:

2352

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask .
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

2353

2354 **Return Values**

2355 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 2356 blocking mode, this indicates that the compatibility tests on di-
 2357 mensions and domains for the input arguments passed successfully.
 2358 Either way, output vector **w** is ready to be used in the next method
 2359 of the sequence.

2360 **GrB_PANIC** Unknown internal error.

2361 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 2362 GraphBLAS objects (input or output) is in an invalid state caused
 2363 by a previous execution error. Call **GrB_error()** to access any error
 2364 messages generated by the implementation.

2365 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

2366 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
 2367 a call to **new** (or **dup** for matrix or vector parameters).

2368 **GrB_DIMENSION_MISMATCH** Mask, vector, and/or matrix dimensions are incompatible.

2369 GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with
 2370 the corresponding domains of the semiring or accumulation opera-
 2371 tor, or the mask's domain is not compatible with `bool` (in the case
 2372 where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

2373 Description

2374 `GrB_m xv` computes the matrix-vector product $w = A \oplus . \otimes u$, or, if an optional binary accumulation
 2375 operator (\odot) is provided, $w = w \odot (A \oplus . \otimes u)$ (where matrix A can be optionally transposed).
 2376 Logically, this operation occurs in three steps:

2377 **Setup** The internal vectors, matrices and mask used in the computation are formed and their
 2378 domains/dimensions are tested for compatibility.

2379 **Compute** The indicated computations are carried out.

2380 **Output** The result is written into the output vector, possibly under control of a mask.

2381 Up to four argument vectors or matrices are used in the `GrB_m xv` operation:

- 2382 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 2383 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 2384 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 2385 4. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

2386 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are
 2387 tested for domain compatibility as follows:

- 2388 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 2389 must be from one of the pre-defined types of Table 2.2.
- 2390 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the semiring.
- 2391 3. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the semiring.
- 2392 4. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the semiring.
- 2393 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 2394 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
 2395 of the accumulation operator.

2396 Two domains are compatible with each other if values from one domain can be cast to values in
 2397 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
 2398 all compatible with each other. A domain from a user-defined type is only compatible with itself.

2399 If any compatibility rule above is violated, execution of `GrB_mvx` ends and the domain mismatch
 2400 error listed above is returned.

2401 From the argument vectors and matrices, the internal matrices and mask used in the computation
 2402 are formed (\leftarrow denotes copy):

- 2403 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 2404 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 2405 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 2406 (b) If `mask \neq GrB_NULL`,
 - 2407 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 2408 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 2409 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 2410 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 2411 4. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

2412 The internal matrices and masks are checked for shape compatibility. The following conditions
 2413 must hold:

- 2414 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$.
- 2415 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 2416 3. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

2417 If any compatibility rule above is violated, execution of `GrB_mvx` ends and the dimension mismatch
 2418 error listed above is returned.

2419 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 2420 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2421 We are now ready to carry out the matrix-vector multiplication and any additional associated
 2422 operations. We describe this in terms of two intermediate vectors:

- 2423 • $\tilde{\mathbf{t}}$: The vector holding the product of matrix $\tilde{\mathbf{A}}$ and vector $\tilde{\mathbf{u}}$.
- 2424 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

2425 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.
 2426 The value of each of its elements is computed by

$$2427 \quad t_i = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

2428 where \oplus and \otimes are the additive and multiplicative operators of semiring `op`, respectively.

2429 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

2430 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

2431 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

2432
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

2433 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
2434 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

2435
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

2436
2437
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

2438
2439
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

2440 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

2441 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
2442 using what is called a *standard vector mask and replace*. This is carried out under control of the
2443 mask which acts as a “write mask”.

2444 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
2445 deleted and the content of the new output vector, \mathbf{w} , is defined as,

2446
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

2447 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
2448 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
2449 mask are unchanged:

2450
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(-\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

2451 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
2452 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
2453 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
2454 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

2455 4.3.4 eWiseMult: Element-wise multiplication

2456 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation
2457 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of
2458 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”
2459 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
2460 the set of values from the resulting index set.

2461 4.3.4.1 eWiseMult: Vector variant

2462 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-
2463 ducing a third vector as result.

2464 C Syntax

```
2465     GrB_Info GrB_eWiseMult(GrB_Vector      w,
2466                           const GrB_Vector  mask,
2467                           const GrB_BinaryOp accum,
2468                           const GrB_Semiring op,
2469                           const GrB_Vector  u,
2470                           const GrB_Vector  v,
2471                           const GrB_Descriptor desc);
2472
2473     GrB_Info GrB_eWiseMult(GrB_Vector      w,
2474                           const GrB_Vector  mask,
2475                           const GrB_BinaryOp accum,
2476                           const GrB_Monoid  op,
2477                           const GrB_Vector  u,
2478                           const GrB_Vector  v,
2479                           const GrB_Descriptor desc);
2480
2481     GrB_Info GrB_eWiseMult(GrB_Vector      w,
2482                           const GrB_Vector  mask,
2483                           const GrB_BinaryOp accum,
2484                           const GrB_BinaryOp op,
2485                           const GrB_Vector  u,
2486                           const GrB_Vector  v,
2487                           const GrB_Descriptor desc);
```

2488 Parameters

2489 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
2490 that may be accumulated with the result of the element-wise operation. On output,
2491 this vector holds the results of the operation.

2492 **mask** (IN) An optional “write” mask that controls which results from this operation are
2493 stored into the output vector **w**. The mask dimensions must match those of the
2494 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
2495 of the mask vector must be of type `bool` or any of the predefined “built-in” types
2496 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
2497 dimensions of **w**), `GrB_NULL` should be specified.

2498 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
2499 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
2500 specified.

2501 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
2502 operation. Depending on which type is passed, the following defines the binary
2503 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

2504
2505
2506
2507
2508

2509
2510

2511
2512

2513
2514
2515

2516

2517

2518
2519
2520
2521
2522

2523

2524
2525
2526
2527

2528

2529
2530

2531

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

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

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 is ignored.

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

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

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

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask .

Return Values

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

GrB_PANIC Unknown internal error.

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

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

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

GrB_DIMENSION_MISMATCH Mask or vector dimensions are incompatible.

2532 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 2533 responding domains of the binary operator (op) or accumulation
 2534 operator, or the mask's domain is not compatible with `bool` (in the
 2535 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

2536 Description

2537 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:
 2538 $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$.
 2539 Logically, this operation occurs in three steps:

2540 **Setup** The internal vectors and mask used in the computation are formed and their domains
 2541 and dimensions are tested for compatibility.

2542 **Compute** The indicated computations are carried out.

2543 **Output** The result is written into the output vector, possibly under control of a mask.

2544 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 2545 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 2546 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 2547 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 2548 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

2549 The argument vectors, the “product” operator (op), and the accumulation operator (if provided)
 2550 are tested for domain compatibility as follows:

- 2551 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 2552 must be from one of the pre-defined types of Table 2.2.
- 2553 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 2554 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 2555 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 2556 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 2557 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 2558 the accumulation operator.

2559 Two domains are compatible with each other if values from one domain can be cast to values
 2560 in the other domain as per the rules of the C language. In particular, domains from Table 2.2
 2561 are all compatible with each other. A domain from a user-defined type is only compatible with

2562 itself. If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the domain
 2563 mismatch error listed above is returned.

2564 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 2565 denotes copy):

- 2566 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 2567 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 2568 (a) If `mask = GrB.NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 2569 (b) If `mask \neq GrB.NULL`,
 - 2570 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 2571 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 2572 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 2573 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 2574 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

2575 The internal vectors and mask are checked for dimension compatibility. The following conditions
 2576 must hold:

- 2577 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}}) = \mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{v}})$.

2578 If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the dimension
 2579 mismatch error listed above is returned.

2580 From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with
 2581 GrB_SUCCESS return code and defer any computation and/or execution error codes.

2582 We are now ready to carry out the element-wise “product” and any additional associated operations.
 2583 We describe this in terms of two intermediate vectors:

- 2584 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “product” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 2585 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

2586 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.
 2587 The value of each of its elements is computed by:

$$2588 \quad t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}))$$

2589 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 2590 • If `accum = GrB.NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

2591 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$2592 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

2593 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 2594 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$2595 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$2596 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$2597 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

2599 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

2601 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 2602 using what is called a *standard vector mask and replace*. This is carried out under control of the
 2603 mask which acts as a “write mask”.

2604 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 2605 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$2606 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

2607 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 2608 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 2609 mask are unchanged:

$$2610 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

2611 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 2612 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 2613 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 2614 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

2615 4.3.4.2 eWiseMult: Matrix variant

2616 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-
 2617 ducing a third matrix as result.

2618 C Syntax

```
2619     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
2620                          const GrB_Matrix Mask,
2621                          const GrB_BinaryOp accum,
2622                          const GrB_Semiring op,
2623                          const GrB_Matrix A,
```

```

2624             const GrB_Matrix      B,
2625             const GrB_Descriptor  desc);
2626
2627 GrB_Info GrB_eWiseMult(GrB_Matrix  C,
2628             const GrB_Matrix      Mask,
2629             const GrB_BinaryOp    accum,
2630             const GrB_Monoid      op,
2631             const GrB_Matrix      A,
2632             const GrB_Matrix      B,
2633             const GrB_Descriptor  desc);
2634
2635 GrB_Info GrB_eWiseMult(GrB_Matrix  C,
2636             const GrB_Matrix      Mask,
2637             const GrB_BinaryOp    accum,
2638             const GrB_BinaryOp    op,
2639             const GrB_Matrix      A,
2640             const GrB_Matrix      B,
2641             const GrB_Descriptor  desc);

```

2642 Parameters

2643 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
2644 that may be accumulated with the result of the element-wise operation. On output,
2645 the matrix holds the results of the operation.

2646 **Mask** (IN) An optional “write” mask that controls which results from this operation are
2647 stored into the output matrix C. The mask dimensions must match those of the
2648 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
2649 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
2650 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
2651 dimensions of C), GrB_NULL should be specified.

2652 **accum** (IN) An optional binary operator used for accumulating entries into existing C
2653 entries. If assignment rather than accumulation is desired, GrB_NULL should be
2654 specified.

2655 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
2656 operation. Depending on which type is passed, the following defines the binary
2657 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

2658 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

2659 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-
2660 nored.

2661 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
2662 is ignored.

2663 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the
2664 operation.

2665 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the
2666 operation.

2667 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
2668 should be specified. Non-default field/value pairs are listed as follows:
2669

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the 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.

2671 Return Values

2672 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
2673 blocking mode, this indicates that the compatibility tests on di-
2674 mensions and domains for the input arguments passed successfully.
2675 Either way, output matrix C is ready to be used in the next method
2676 of the sequence.

2677 GrB_PANIC Unknown internal error.

2678 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
2679 GraphBLAS objects (input or output) is in an invalid state caused
2680 by a previous execution error. Call GrB_error() to access any error
2681 messages generated by the implementation.

2682 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

2683 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
2684 a call to new (or Matrix_dup for matrix parameters).

2685 GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

2686 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
2687 corresponding domains of the binary operator (op) or accumulation
2688 operator, or the mask's domain is not compatible with bool (in the
2689 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

2690 **Description**

2691 This variant of GrB_eWiseMult computes the element-wise “product” of two GraphBLAS matrices:
2692 $C = A \otimes B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \otimes B)$.
2693 Logically, this operation occurs in three steps:

2694 **Setup** The internal matrices and mask used in the computation are formed and their domains
2695 and dimensions are tested for compatibility.

2696 **Compute** The indicated computations are carried out.

2697 **Output** The result is written into the output matrix, possibly under control of a mask.

2698 Up to four argument matrices are used in the GrB_eWiseMult operation:

- 2699 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 2700 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)
- 2701 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 2702 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

2703 The argument matrices, the “product” operator (op), and the accumulation operator (if provided)
2704 are tested for domain compatibility as follows:

- 2705 1. If Mask is not GrB_NULL, and $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is not set, then $\mathbf{D}(\text{Mask})$
2706 must be from one of the pre-defined types of Table 2.2.
- 2707 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 2708 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 2709 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 2710 5. If accum is not GrB_NULL, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
2711 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of op must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
2712 the accumulation operator.

2713 Two domains are compatible with each other if values from one domain can be cast to values
2714 in the other domain as per the rules of the C language. In particular, domains from Table 2.2
2715 are all compatible with each other. A domain from a user-defined type is only compatible with
2716 itself. If any compatibility rule above is violated, execution of GrB_eWiseMult ends and the domain
2717 mismatch error listed above is returned.

2718 From the argument matrices, the internal matrices and mask used in the computation are formed
2719 (\leftarrow denotes copy):

- 2720 1. Matrix $\tilde{C} \leftarrow C$.

- 2721 2. Two-dimensional mask, $\widetilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
- 2722 (a) If `Mask = GrB_NULL`, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
2723 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 2724 (b) If `Mask \neq GrB_NULL`,
- 2725 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
2726 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
- 2727 ii. Otherwise, $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
2728 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
- 2729 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$.
- 2730 3. Matrix $\widetilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 2731 4. Matrix $\widetilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

2732 The internal matrices and masks are checked for dimension compatibility. The following conditions
2733 must hold:

- 2734 1. $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}}) = \mathbf{nrows}(\widetilde{\mathbf{A}}) = \mathbf{nrows}(\widetilde{\mathbf{B}})$.
- 2735 2. $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}}) = \mathbf{ncols}(\widetilde{\mathbf{A}}) = \mathbf{ncols}(\widetilde{\mathbf{B}})$.

2736 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
2737 mismatch error listed above is returned.

2738 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
2739 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2740 We are now ready to carry out the element-wise “product” and any additional associated operations.
2741 We describe this in terms of two intermediate matrices:

- 2742 • $\widetilde{\mathbf{T}}$: The matrix holding the element-wise product of $\widetilde{\mathbf{A}}$ and $\widetilde{\mathbf{B}}$.
- 2743 • $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

2744 The intermediate matrix $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}}) \neq \emptyset\} \rangle$
2745 is created. The value of each of its elements is computed by

$$2746 T_{ij} = (\widetilde{\mathbf{A}}(i, j) \otimes \widetilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}})$$

2747 The intermediate matrix $\widetilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 2748 • If `accum = GrB_NULL`, then $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$.
- 2749 • If `accum` is a binary operator, then $\widetilde{\mathbf{Z}}$ is defined as

$$2750 \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

2751 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 2752 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$2753 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$2754 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$2755 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

2756 where $\odot = \bigcirc(\text{accum})$, and the difference operator refers to set difference.
 2757

2758 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 2759 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 2760 mask which acts as a “write mask”.
 2761

- 2762 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{C} on input to this operation are
 2763 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$2764 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 2765 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 2766 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 2767 mask are unchanged:

$$2768 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

2769 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
 2770 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
 2771 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but
 2772 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 2773 sequence.

2774 4.3.5 eWiseAdd: Element-wise addition

2775 **Note:** The difference between eWiseAdd and eWiseMult is not about the element-wise operation
 2776 but how the index sets are treated. eWiseAdd returns an object whose indices are the “union” of
 2777 the indices of the inputs whereas eWiseMult returns an object whose indices are the “intersection”
 2778 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on
 2779 the set of values from the resulting index set.

2780 4.3.5.1 eWiseAdd: Vector variant

2781 Perform element-wise (general) addition on the elements of two vectors, producing a third vector
 2782 as result.

2783 C Syntax

```
2784     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
2785                          const GrB_Vector mask,  
2786                          const GrB_BinaryOp accum,  
2787                          const GrB_Semiring op,  
2788                          const GrB_Vector u,  
2789                          const GrB_Vector v,  
2790                          const GrB_Descriptor desc);  
2791  
2792     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
2793                          const GrB_Vector mask,  
2794                          const GrB_BinaryOp accum,  
2795                          const GrB_Monoid op,  
2796                          const GrB_Vector u,  
2797                          const GrB_Vector v,  
2798                          const GrB_Descriptor desc);  
2799  
2800     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
2801                          const GrB_Vector mask,  
2802                          const GrB_BinaryOp accum,  
2803                          const GrB_BinaryOp op,  
2804                          const GrB_Vector u,  
2805                          const GrB_Vector v,  
2806                          const GrB_Descriptor desc);
```

2807 Parameters

2808 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
2809 that may be accumulated with the result of the element-wise operation. On output,
2810 this vector holds the results of the operation.

2811 **mask** (IN) An optional “write” mask that controls which results from this operation are
2812 stored into the output vector **w**. The mask dimensions must match those of the
2813 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
2814 of the mask vector must be of type `bool` or any of the predefined “built-in” types
2815 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
2816 dimensions of **w**), `GrB_NULL` should be specified.

2817 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
2818 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
2819 specified.

2820 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
2821 operation. Depending on which type is passed, the following defines the binary
2822 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

2823
2824
2825
2826
2827

2828
2829

2830
2831

2832
2833
2834

2835

2836

2837
2838
2839
2840
2841

2842

2843
2844
2845
2846

2847

2848
2849

2850

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

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

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 binary op and additive identity are ignored.

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

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

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

Param	Field	Value	Description
<code>w</code>	<code>GrB_OUTP</code>	<code>GrB_REPLACE</code>	Output vector <code>w</code> is cleared (all elements removed) before the result is stored in it.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_STRUCTURE</code>	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
<code>mask</code>	<code>GrB_MASK</code>	<code>GrB_COMP</code>	Use the complement of <code>mask</code> .

Return Values

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

`GrB_PANIC` Unknown internal error.

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

`GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

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

`GrB_DIMENSION_MISMATCH` Mask or vector dimensions are incompatible.

2851 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the cor-
 2852 responding domains of the binary operator (op) or accumulation
 2853 operator, or the mask’s domain is not compatible with `bool` (in the
 2854 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

2855 **Description**

2856 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors: $\mathbf{w} =$
 2857 $\mathbf{u} \oplus \mathbf{v}$, or, if an optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$. Logically,
 2858 this operation occurs in three steps:

2859 **Setup** The internal vectors and mask used in the computation are formed and their domains
 2860 and dimensions are tested for compatibility.

2861 **Compute** The indicated computations are carried out.

2862 **Output** The result is written into the output vector, possibly under control of a mask.

2863 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 2864 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 2865 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 2866 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 2867 4. $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

2868 The argument vectors, the “sum” operator (op), and the accumulation operator (if provided) are
 2869 tested for domain compatibility as follows:

- 2870 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
 2871 must be from one of the pre-defined types of Table 2.2.
- 2872 2. $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 2873 3. $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 2874 4. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 2875 5. $\mathbf{D}(\mathbf{u})$ and $\mathbf{D}(\mathbf{v})$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 2876 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 2877 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 2878 the accumulation operator.

2879 Two domains are compatible with each other if values from one domain can be cast to values
 2880 in the other domain as per the rules of the C language. In particular, domains from Table 2.2
 2881 are all compatible with each other. A domain from a user-defined type is only compatible with
 2882 itself. If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain
 2883 mismatch error listed above is returned.

2884 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 2885 denotes copy):

- 2886 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 2887 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 2888 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 2889 (b) If `mask \neq GrB_NULL`,
 - 2890 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 2891 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 2892 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 2893 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.
- 2894 4. Vector $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$.

2895 The internal vectors and mask are checked for dimension compatibility. The following conditions
 2896 must hold:

- 2897 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}}) = \mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{v}})$.

2898 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 2899 mismatch error listed above is returned.

2900 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 2901 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

2902 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 2903 We describe this in terms of two intermediate vectors:

- 2904 • $\tilde{\mathbf{t}}$: The vector holding the element-wise “sum” of $\tilde{\mathbf{u}}$ and vector $\tilde{\mathbf{v}}$.
- 2905 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

2906 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.
 2907 The value of each of its elements is computed by:

$$2908 \quad t_i = (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}}))$$

$$2909 \quad t_i = \tilde{\mathbf{u}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{u}}) - (\mathbf{ind}(\tilde{\mathbf{v}}) \cap \mathbf{ind}(\tilde{\mathbf{u}})))$$

2911

2912

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

2913

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

2914

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

2915

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

2916

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

2917

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

2918

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

2919

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

2920

2921

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

2922

2923

2924

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

2925

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

2926

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

2927

2928

2929

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

2930

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

2931

2932

- 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:

2933

2934

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

2935

2936

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.

2937

2938

2939

2940

4.3.5.2 eWiseAdd: Matrix variant

2941

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

2942

2943 C Syntax

```
2944     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,  
2945                          const GrB_Matrix Mask,  
2946                          const GrB_BinaryOp accum,  
2947                          const GrB_Semiring op,  
2948                          const GrB_Matrix A,  
2949                          const GrB_Matrix B,  
2950                          const GrB_Descriptor desc);  
2951  
2952     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,  
2953                          const GrB_Matrix Mask,  
2954                          const GrB_BinaryOp accum,  
2955                          const GrB_Monoid op,  
2956                          const GrB_Matrix A,  
2957                          const GrB_Matrix B,  
2958                          const GrB_Descriptor desc);  
2959  
2960     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,  
2961                          const GrB_Matrix Mask,  
2962                          const GrB_BinaryOp accum,  
2963                          const GrB_BinaryOp op,  
2964                          const GrB_Matrix A,  
2965                          const GrB_Matrix B,  
2966                          const GrB_Descriptor desc);
```

2967 Parameters

2968 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
2969 that may be accumulated with the result of the element-wise operation. On output,
2970 the matrix holds the results of the operation.

2971 **Mask** (IN) An optional “write” mask that controls which results from this operation are
2972 stored into the output matrix C. The mask dimensions must match those of the
2973 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
2974 of the Mask matrix must be of type bool or any of the predefined “built-in” types
2975 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
2976 dimensions of C), GrB_NULL should be specified.

2977 **accum** (IN) An optional binary operator used for accumulating entries into existing C
2978 entries. If assignment rather than accumulation is desired, GrB_NULL should be
2979 specified.

2980 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”
2981 operation. Depending on which type is passed, the following defines the binary
2982 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$, used:

2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994

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

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

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 binary op and additive identity are ignored.

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

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

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

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the 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.

2995

2996 Return Values

2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010

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

GrB_PANIC Unknown internal error.

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

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

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

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

3011 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 3012 corresponding domains of the binary operator (\oplus) or accumulation
 3013 operator, or the mask's domain is not compatible with `bool` (in the
 3014 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3015 Description

3016 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:
 3017 $C = A \oplus B$, or, if an optional binary accumulation operator (\odot) is provided, $C = C \odot (A \oplus B)$.
 3018 Logically, this operation occurs in three steps:

3019 **Setup** The internal matrices and mask used in the computation are formed and their domains
 3020 and dimensions are tested for compatibility.

3021 **Compute** The indicated computations are carried out.

3022 **Output** The result is written into the output matrix, possibly under control of a mask.

3023 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3024 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3025 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)
- 3026 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3027 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3028 The argument matrices, the “sum” operator (\oplus), and the accumulation operator (if provided) are
 3029 tested for domain compatibility as follows:

- 3030 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3031 must be from one of the pre-defined types of Table 2.2.
- 3032 2. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\oplus)$.
- 3033 3. $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{in_2}(\oplus)$.
- 3034 4. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\oplus)$.
- 3035 5. $\mathbf{D}(A)$ and $\mathbf{D}(B)$ must be compatible with $\mathbf{D}_{out}(\oplus)$.
- 3036 6. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3037 of the accumulation operator and $\mathbf{D}_{out}(\oplus)$ of \oplus must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
 3038 the accumulation operator.

3039 Two domains are compatible with each other if values from one domain can be cast to values
 3040 in the other domain as per the rules of the C language. In particular, domains from Table 2.2
 3041 are all compatible with each other. A domain from a user-defined type is only compatible with
 3042 itself. If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain
 3043 mismatch error listed above is returned.

3044 From the argument matrices, the internal matrices and mask used in the computation are formed
 3045 (\leftarrow denotes copy):

- 3046 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3047 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 3048 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 3049 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 3050 (b) If `Mask \neq GrB_NULL`,
 - 3051 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 3052 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 3053 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 3054 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 3055 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3056 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3057 4. Matrix $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

3058 The internal matrices and masks are checked for dimension compatibility. The following conditions
 3059 must hold:

- 3060 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 3061 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$.

3062 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension
 3063 mismatch error listed above is returned.

3064 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 3065 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3066 We are now ready to carry out the element-wise “sum” and any additional associated operations.
 3067 We describe this in terms of two intermediate matrices:

- 3068 • $\tilde{\mathbf{T}}$: The matrix holding the element-wise sum of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 3069 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3070 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$
 3071 is created. The value of each of its elements is computed by

$$3072 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

$$3073 \quad T_{ij} = \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{B}}) \cap \mathbf{ind}(\tilde{\mathbf{A}})))$$

$$3074 \quad T_{ij} = \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{B}}) \cap \mathbf{ind}(\tilde{\mathbf{A}})))$$

3077 where the difference operator in the previous expressions refers to set difference.

3078 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3079 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 3080 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3081 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3082 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3083 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3084 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3085 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3086 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3089 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3090 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 3091 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 3092 mask which acts as a “write mask”.

- 3093 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 3094 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3095 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3096 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
 3097 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
 3098 mask are unchanged:

$$3099 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3100 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 3101 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 3102 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
 3103 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
 3104 sequence.

3105 4.3.6 extract: Selecting Sub-Graphs

3106 Extract a subset of a matrix or vector.

3107 4.3.6.1 extract: Standard vector variant

3108 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector
3109 whose size is equal to the number of indices.

3110 C Syntax

```
3111     GrB_Info GrB_extract(GrB_Vector      w,  
3112                        const GrB_Vector mask,  
3113                        const GrB_BinaryOp accum,  
3114                        const GrB_Vector  u,  
3115                        const GrB_Index  *indices,  
3116                        GrB_Index      nindices,  
3117                        const GrB_Descriptor desc);
```

3118 Parameters

3119 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3120 that may be accumulated with the result of the extract operation. On output, this
3121 vector holds the results of the operation.

3122 **mask** (IN) An optional “write” mask that controls which results from this operation are
3123 stored into the output vector **w**. The mask dimensions must match those of the
3124 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3125 of the **mask** vector must be of type `bool` or any of the predefined “built-in” types
3126 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
3127 dimensions of **w**), `GrB_NULL` should be specified.

3128 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3129 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3130 specified.

3131 **u** (IN) The GraphBLAS vector from which the subset is extracted.

3132 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of
3133 elements from **u** that are extracted. If all elements of **u** are to be extracted in order
3134 from 0 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution
3135 mode and return value, this array may be manipulated by the caller after this
3136 operation returns without affecting any deferred computations for this operation.

3137 **nindices** (IN) The number of values in **indices** array. Must be equal to `size(w)`.

3138
3139
3140

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:

3141

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

3142 Return Values

3143
3144
3145
3146
3147

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.

3148

GrB_PANIC Unknown internal error.

3149
3150
3151
3152

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.

3153

GrB_OUT_OF_MEMORY Not enough memory available for operation.

3154
3155

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

3156
3157

GrB_INDEX_OUT_OF_BOUNDS A value in indices is greater than or equal to size(u). In non-blocking mode, this error can be deferred.

3158

GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or nindices \neq size(w).

3159
3160
3161
3162

GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

3163

GrB_NULL_POINTER Argument row_indices is a NULL pointer.

3164 Description

3165
3166

This variant of GrB_extract computes the result of extracting a subset of locations from a GraphBLAS vector in a specific order: $w = u(\text{indices})$; or, if an optional binary accumulation operator

3167 (\odot) is provided, $w = w \odot u(\text{indices})$. More explicitly:

$$\begin{aligned} 3168 \quad w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

3169 Logically, this operation occurs in three steps:

3170 **Setup** The internal vectors and mask used in the computation are formed and their domains
3171 and dimensions are tested for compatibility.

3172 **Compute** The indicated computations are carried out.

3173 **Output** The result is written into the output vector, possibly under control of a mask.

3174 Up to three argument vectors are used in this `GrB_extract` operation:

- 3175 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3176 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3177 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3178 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
3179 bility as follows:

- 3180 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3181 must be from one of the pre-defined types of Table 2.2.
- 3182 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(u)$.
- 3183 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3184 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
3185 mulation operator.

3186 Two domains are compatible with each other if values from one domain can be cast to values in
3187 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
3188 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3189 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
3190 error listed above is returned.

3191 From the arguments, the internal vectors, `mask`, and index array used in the computation are
3192 formed (\leftarrow denotes copy):

- 3193 1. Vector $\tilde{w} \leftarrow w$.
- 3194 2. One-dimensional mask, \tilde{m} , is computed from argument `mask` as follows:
 - 3195 (a) If `mask = GrB_NULL`, then $\tilde{m} = \langle \mathbf{size}(w), \{i, \forall i : 0 \leq i < \mathbf{size}(w)\} \rangle$.

- 3196 (b) If $\text{mask} \neq \text{GrB_NULL}$,
- 3197 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
- 3198 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 3199 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.

3200 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3201 4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:

- 3202 (a) If $\text{indices} = \text{GrB_ALL}$, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.
- 3203 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

3204 The internal vectors and mask are checked for dimension compatibility. The following conditions
3205 must hold:

- 3206 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 3207 2. $\text{nindices} = \text{size}(\tilde{\mathbf{w}})$.

3208 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
3209 match error listed above is returned.

3210 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3211 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3212 We are now ready to carry out the extract and any additional associated operations. We describe
3213 this in terms of two intermediate vectors:

- 3214 • $\tilde{\mathbf{t}}$: The vector holding the extraction from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.
- 3215 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3216 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$3217 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(i, \tilde{\mathbf{u}}[\tilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \tilde{\mathbf{I}}[i] \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

3218 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
3219 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`
3220 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
3221 result vector, \mathbf{w} , is invalid from this point forward in the sequence.

3222 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 3223 • If $\text{accum} = \text{GrB_NULL}$, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 3224 • If accum is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3225 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3226 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3227 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned}
 3228 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\
 3229 \\
 3230 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 3231 \\
 3232 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),
 \end{aligned}$$

3233 where $\odot = \bigodot(\text{accum})$, and the difference operator refers to set difference.

3234 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3235 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3236 mask which acts as a “write mask”.

- 3237 • If desc[GrB_OUTP].GrB_REPLACE is set, then any values in \mathbf{w} on input to this operation are
 3238 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3239 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3240 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3241 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3242 mask are unchanged:

$$3243 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3244 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content of
 3245 vector \mathbf{w} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method exits
 3246 with return value GrB_SUCCESS and the new content of vector \mathbf{w} is as defined above but may not
 3247 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

3248 4.3.6.2 extract: Standard matrix variant

3249 Extract a sub-matrix from a larger matrix as specified by a set of row indices and a set of column
 3250 indices. The result is a matrix whose size is equal to size of the sets of indices.

3251 C Syntax

```

3252     GrB_Info GrB_extract(GrB_Matrix      C,
3253                        const GrB_Matrix  Mask,
3254                        const GrB_BinaryOp accum,
3255                        const GrB_Matrix  A,
3256                        const GrB_Index   *row_indices,
3257                        GrB_Index         nrows,
3258                        const GrB_Index   *col_indices,
3259                        GrB_Index         ncols,
3260                        const GrB_Descriptor desc);
  
```

3261 **Parameters**

3262 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 3263 that may be accumulated with the result of the extract operation. On output, the
 3264 matrix holds the results of the operation.

3265 Mask (IN) An optional “write” mask that controls which results from this operation are
 3266 stored into the output matrix C. The mask dimensions must match those of the
 3267 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 3268 of the Mask matrix must be of type bool or any of the predefined “built-in” types
 3269 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
 3270 dimensions of C), GrB_NULL should be specified.

3271 accum (IN) An optional binary operator used for accumulating entries into existing C
 3272 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 3273 specified.

3274 A (IN) The GraphBLAS matrix from which the subset is extracted.

3275 row_indices (IN) Pointer to the ordered set (array) of indices corresponding to the rows of A
 3276 from which elements are extracted. If elements in all rows of A are to be extracted
 3277 in order, GrB_ALL should be specified. Regardless of execution mode and return
 3278 value, this array may be manipulated by the caller after this operation returns
 3279 without affecting any deferred computations for this operation.

3280 nrows (IN) The number of values in the row_indices array. Must be equal to **nrows(C)**.

3281 col_indices (IN) Pointer to the ordered set (array) of indices corresponding to the columns
 3282 of A from which elements are extracted. If elements in all columns of A are to
 3283 be extracted in order, then GrB_ALL should be specified. Regardless of execution
 3284 mode and return value, this array may be manipulated by the caller after this
 3285 operation returns without affecting any deferred computations for this operation.

3286 ncols (IN) The number of values in the col_indices array. Must be equal to **ncols(C)**.

3287 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3288 should be specified. Non-default field/value pairs are listed as follows:
 3289

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

3290

3291 Return Values

3292 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3293 blocking mode, this indicates that the compatibility tests on di-
3294 mensions and domains for the input arguments passed successfully.
3295 Either way, output matrix C is ready to be used in the next method
3296 of the sequence.

3297 GrB_PANIC Unknown internal error.

3298 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3299 GraphBLAS objects (input or output) is in an invalid state caused
3300 by a previous execution error. Call GrB_error() to access any error
3301 messages generated by the implementation.

3302 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3303 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3304 a call to new (or Matrix_dup for matrix parameters).

3305 GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to nrows(A), or a
3306 value in col_indices is greater than or equal to ncols(A). In non-
3307 blocking mode, this error can be deferred.

3308 GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, nrows \neq nrows(C), or
3309 ncols \neq ncols(C).

3310 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with each
3311 other or the corresponding domains of the accumulation operator,
3312 or the mask's domain is not compatible with bool (in the case where
3313 desc[GrB_MASK].GrB_STRUCTURE is not set).

3314 GrB_NULL_POINTER Either argument row_indices is a NULL pointer, argument col_indices
3315 is a NULL pointer, or both.

3316 Description

3317 This variant of GrB_extract computes the result of extracting a subset of locations from specified
3318 rows and columns of a GraphBLAS matrix in a specific order: $C = A(\text{row_indices}, \text{col_indices})$; or, if
3319 an optional binary accumulation operator (\odot) is provided, $C = C \odot A(\text{row_indices}, \text{col_indices})$. More
3320 explicitly (not accounting for an optional transpose of A):

$$\begin{aligned} 3321 \quad C(i, j) &= A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ C(i, j) &= C(i, j) \odot A(\text{row_indices}[i], \text{col_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

3322 Logically, this operation occurs in three steps:

3323 **Setup** The internal matrices and mask used in the computation are formed and their domains
3324 and dimensions are tested for compatibility.

3325 **Compute** The indicated computations are carried out.

3326 **Output** The result is written into the output matrix, possibly under control of a mask.

3327 Up to three argument matrices are used in the `GrB_extract` operation:

- 3328 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3329 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)
- 3330 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

3331 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
3332 ibility as follows:

- 3333 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
3334 must be from one of the pre-defined types of Table 2.2.
- 3335 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 3336 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3337 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
3338 mulation operator.

3339 Two domains are compatible with each other if values from one domain can be cast to values in
3340 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
3341 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3342 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
3343 error listed above is returned.

3344 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are
3345 formed (\leftarrow denotes copy):

- 3346 1. Matrix $\tilde{C} \leftarrow C$.
- 3347 2. Two-dimensional mask, \tilde{M} , is computed from argument `Mask` as follows:
 - 3348 (a) If `Mask` = `GrB_NULL`, then $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$
3349 $j < \mathbf{ncols}(C)\} \rangle$.
 - 3350 (b) If `Mask` \neq `GrB_NULL`,
 - 3351 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
3352 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
 - 3353 ii. Otherwise, $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
3354 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
 - 3355 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{M} \leftarrow \neg \tilde{M}$.
- 3356 3. Matrix $\tilde{A} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? A^T : A$.

- 3357 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 3358 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 3359 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i], \forall i : 0 \leq i < \text{nrows}$.
- 3360 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 3361 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$.
- 3362 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \text{ncols}$.

3363 The internal matrices and mask are checked for dimension compatibility. The following conditions
3364 must hold:

- 3365 1. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$.
- 3366 2. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$.
- 3367 3. $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$.
- 3368 4. $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$.

3369 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
3370 match error listed above is returned.

3371 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3372 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3373 We are now ready to carry out the extract and any additional associated operations. We describe
3374 this in terms of two intermediate matrices:

- 3375 • $\tilde{\mathbf{T}}$: The matrix holding the extraction from $\tilde{\mathbf{A}}$.
- 3376 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3377 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

3378
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{(i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

3379 At this point, if any value in the $\tilde{\mathbf{I}}$ array is not in the range $[0, \text{nrows}(\tilde{\mathbf{A}}))$ or any value in the $\tilde{\mathbf{J}}$
3380 array is not in the range $[0, \text{ncols}(\tilde{\mathbf{A}}))$, the execution of `GrB_extract` ends and the index out-of-
3381 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
3382 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
3383 this point forward in the sequence.

3384 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 3385 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

3386 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3387 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3388 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
3389 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3390 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

3391

$$3392 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3393

$$3394 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3395 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3396 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
3397 using what is called a *standard matrix mask and replace*. This is carried out under control of the
3398 mask which acts as a “write mask”.

3399 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
3400 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$3401 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3402 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
3403 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
3404 mask are unchanged:

$$3405 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3406 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
3407 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
3408 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
3409 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
3410 sequence.

3411 4.3.6.3 extract: Column (and row) variant

3412 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the
3413 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as
3414 well.

3415 C Syntax

```
3416     GrB_Info GrB_extract(GrB_Vector      w,  
3417                         const GrB_Vector mask,  
3418                         const GrB_BinaryOp accum,  
3419                         const GrB_Matrix  A,  
3420                         const GrB_Index  *row_indices,  
3421                         GrB_Index       nrows,  
3422                         GrB_Index       col_index,  
3423                         const GrB_Descriptor desc);
```

3424 Parameters

3425 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3426 that may be accumulated with the result of the extract operation. On output, this
3427 vector holds the results of the operation.

3428 **mask** (IN) An optional “write” mask that controls which results from this operation are
3429 stored into the output vector **w**. The mask dimensions must match those of the
3430 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3431 of the **mask** vector must be of type `bool` or any of the predefined “built-in” types
3432 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
3433 dimensions of **w**), `GrB_NULL` should be specified.

3434 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3435 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3436 specified.

3437 **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

3438 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations
3439 within the specified column of **A** from which elements are extracted. If elements in
3440 all rows of **A** are to be extracted in order, `GrB_ALL` should be specified. Regardless
3441 of execution mode and return value, this array may be manipulated by the caller
3442 after this operation returns without affecting any deferred computations for this
3443 operation.

3444 **nrows** (IN) The number of indices in the **row_indices** array. Must be equal to `size(w)`.

3445 **col_index** (IN) The index of the column of **A** from which to extract values. It must be in the
3446 range `[0, ncols(A))`.

3447 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
3448 should be specified. Non-default field/value pairs are listed as follows:
3449

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

3450

3451 Return Values

3452 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3453 blocking mode, this indicates that the compatibility tests on di-
 3454 mensions and domains for the input arguments passed successfully.
 3455 Either way, output vector w is ready to be used in the next method
 3456 of the sequence.

3457 GrB_PANIC Unknown internal error.

3458 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 3459 GraphBLAS objects (input or output) is in an invalid state caused
 3460 by a previous execution error. Call GrB_error() to access any error
 3461 messages generated by the implementation.

3462 GrB_OUT_OF_MEMORY Not enough memory available for operation.

3463 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
 3464 a call to new (or dup for vector or matrix parameters).

3465 GrB_INVALID_INDEX col_index is outside the allowable range (i.e., greater than **ncols(A)**).

3466 GrB_INDEX_OUT_OF_BOUNDS A value in row_indices is greater than or equal to **nrows(A)**. In
 3467 non-blocking mode, this error can be deferred.

3468 GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or **nrows** \neq **size(w)**.

3469 GrB_DOMAIN_MISMATCH The domains of the vector or matrix are incompatible with each
 3470 other or the corresponding domains of the accumulation operator,
 3471 or the mask's domain is not compatible with **bool** (in the case where
 3472 desc[GrB_MASK].GrB_STRUCTURE is not set).

3473 GrB_NULL_POINTER Argument row_indices is a NULL pointer.

3474 Description

3475 This variant of GrB_extract computes the result of extracting a subset of locations (in a specific
 3476 order) from a specified column of a GraphBLAS matrix: $w = A(:, col_index)(row_indices)$; or, if an

3477 optional binary accumulation operator (\odot) is provided, $\mathbf{w} = \mathbf{w} \odot \mathbf{A}(:, \text{col_index})(\text{row_indices})$. More
 3478 explicitly:

$$3479 \quad \begin{aligned} \mathbf{w}(i) &= \mathbf{A}(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ \mathbf{w}(i) &= \mathbf{w}(i) \odot \mathbf{A}(\text{row_indices}[i], \text{col_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

3480 Logically, this operation occurs in three steps:

3481 **Setup** The internal matrices, vectors, and mask used in the computation are formed and their
 3482 domains and dimensions are tested for compatibility.

3483 **Compute** The indicated computations are carried out.

3484 **Output** The result is written into the output vector, possibly under control of a mask.

3485 Up to three argument vectors and matrices are used in this `GrB_extract` operation:

- 3486 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3487 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3488 3. $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3489 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain
 3490 compatibility as follows:

- 3491 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 3492 must be from one of the pre-defined types of Table 2.2.
- 3493 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{A})$.
- 3494 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3495 of the accumulation operator and $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 3496 mulation operator.

3497 Two domains are compatible with each other if values from one domain can be cast to values in
 3498 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 3499 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3500 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch
 3501 error listed above is returned.

3502 From the arguments, the internal vector, matrix, mask, and index array used in the computation
 3503 are formed (\leftarrow denotes copy):

- 3504 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3505 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3506 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.

- 3507 (b) If $\text{mask} \neq \text{GrB_NULL}$,
- 3508 i. If $\text{desc}[\text{GrB_MASK}].\text{GrB_STRUCTURE}$ is set, then $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$,
- 3509 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.
- 3510 (c) If $\text{desc}[\text{GrB_MASK}].\text{GrB_COMP}$ is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.
- 3511 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3512 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 3513 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$.
- 3514 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$.

3515 The internal vector, `mask`, and index array are checked for dimension compatibility. The following
3516 conditions must hold:

- 3517 1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 3518 2. $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}$.

3519 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-
3520 match error listed above is returned.

3521 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 3522 1. $0 \leq \text{col_index} < \text{ncols}(\mathbf{A})$

3523 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above
3524 is returned.

3525 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3526 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3527 We are now ready to carry out the extract and any additional associated operations. We describe
3528 this in terms of two intermediate vectors:

- 3529 • $\tilde{\mathbf{t}}$: The vector holding the extraction from a column of $\tilde{\mathbf{A}}$.
- 3530 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

3531 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

3532
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \text{col_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\tilde{\mathbf{I}}[i], \text{col_index}) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

3533 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`
3534 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,
3535 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result
3536 vector, `w`, is invalid from this point forward in the sequence.

3537 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

3538 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

3539 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

3540
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3541 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
3542 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

3543
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3544

3545
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3546

3547
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3548 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3549 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
3550 using what is called a *standard vector mask and replace*. This is carried out under control of the
3551 mask which acts as a “write mask”.

3552 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
3553 deleted and the content of the new output vector, \mathbf{w} , is defined as,

3554
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3555 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
3556 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
3557 mask are unchanged:

3558
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3559 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
3560 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
3561 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
3562 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

3563 4.3.7 assign: Modifying Sub-Graphs

3564 Assign the contents of a subset of a matrix or vector.

3565 4.3.7.1 assign: Standard vector variant

3566 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.
3567 The size of the input vector is the same size as the index array provided.

3568 **C Syntax**

```
3569         GrB_Info GrB_assign(GrB_Vector          w,  
3570                             const GrB_Vector    mask,  
3571                             const GrB_BinaryOp   accum,  
3572                             const GrB_Vector    u,  
3573                             const GrB_Index      *indices,  
3574                             GrB_Index          nindices,  
3575                             const GrB_Descriptor desc);
```

3576 **Parameters**

3577 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
3578 that may be accumulated with the result of the assign operation. On output, this
3579 vector holds the results of the operation.

3580 **mask** (IN) An optional “write” mask that controls which results from this operation are
3581 stored into the output vector **w**. The mask dimensions must match those of the
3582 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
3583 of the **mask** vector must be of type `bool` or any of the predefined “built-in” types
3584 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
3585 dimensions of **w**), `GrB_NULL` should be specified.

3586 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
3587 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
3588 specified.

3589 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

3590 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
3591 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0
3592 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode
3593 and return value, this array may be manipulated by the caller after this operation
3594 returns without affecting any deferred computations for this operation. If this
3595 array contains duplicate values, it implies in assignment of more than one value to
3596 the same location which leads to undefined results.

3597 **nindices** (IN) The number of values in **indices** array. Must be equal to `size(u)`.

3598 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
3599 should be specified. Non-default field/value pairs are listed as follows:
3600

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

3601

3602 Return Values

3603 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
3604 blocking mode, this indicates that the compatibility tests on di-
3605 mensions and domains for the input arguments passed successfully.
3606 Either way, output vector w is ready to be used in the next method
3607 of the sequence.

3608 GrB_PANIC Unknown internal error.

3609 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3610 GraphBLAS objects (input or output) is in an invalid state caused
3611 by a previous execution error. Call GrB_error() to access any error
3612 messages generated by the implementation.

3613 GrB_OUT_OF_MEMORY Not enough memory available for operation.

3614 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3615 a call to new (or dup for vector parameters).

3616 GrB_INDEX_OUT_OF_BOUNDS A value in indices is greater than or equal to size(w). In non-
3617 blocking mode, this can be reported as an execution error.

3618 GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or nindices \neq size(u).

3619 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with each
3620 other or the corresponding domains of the accumulation operator,
3621 or the mask's domain is not compatible with bool (in the case where
3622 desc[GrB_MASK].GrB_STRUCTURE is not set).

3623 GrB_NULL_POINTER Argument indices is a NULL pointer.

3624 Description

3625 This variant of GrB_assign computes the result of assigning elements from a source GraphBLAS
3626 vector to a destination GraphBLAS vector in a specific order: $w(\text{indices}) = u$; or, if an optional
3627 binary accumulation operator (\odot) is provided, $w(\text{indices}) = w(\text{indices}) \odot u$. More explicitly:

$$\begin{aligned}
 3628 \quad w(\text{indices}[i]) &= u(i), \quad \forall i : 0 \leq i < n\text{indices}, \quad \text{or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \quad \forall i : 0 \leq i < n\text{indices}.
 \end{aligned}$$

3629 Logically, this operation occurs in three steps:

3630 **Setup** The internal vectors and mask used in the computation are formed and their domains
3631 and dimensions are tested for compatibility.

3632 **Compute** The indicated computations are carried out.

3633 **Output** The result is written into the output vector, possibly under control of a mask.

3634 Up to three argument vectors are used in the `GrB_assign` operation:

- 3635 1. $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3636 2. $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 3637 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

3638 The argument vectors and the accumulation operator (if provided) are tested for domain compati-
3639 bility as follows:

- 3640 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\mathbf{mask})$
3641 must be from one of the pre-defined types of Table 2.2.
- 3642 2. $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}(\mathbf{u})$.
- 3643 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathbf{accum})$ and $\mathbf{D}_{out}(\mathbf{accum})$
3644 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\mathbf{accum})$ of the accu-
3645 mulation operator.

3646 Two domains are compatible with each other if values from one domain can be cast to values in
3647 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
3648 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3649 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
3650 error listed above is returned.

3651 From the arguments, the internal vectors, mask and index array used in the computation are formed
3652 (\leftarrow denotes copy):

- 3653 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 3654 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 3655 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 3656 (b) If `mask` \neq `GrB_NULL`,
 - 3657 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 3658 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 3659 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

3660

3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

3661

4. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:

3662

(a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$.

3663

(b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$.

3664

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

3665

3666

1. $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

3667

2. $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$.

3668

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

3669

3670

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.

3671

3672

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

3673

3674

- $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{w}}$.

3675

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

3676

The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

3677

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

3678

At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{w}}$, computation ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result vector, \mathbf{w} , is invalid from this point forward in the sequence.

3679

3680

3681

3682

The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

3683

- If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

3684

$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3685

The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure of $\tilde{\mathbf{w}}$ ($\text{ind}(\tilde{\mathbf{w}})$) and remove from it all the indices of $\tilde{\mathbf{w}}$ that are in the set of indices being assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

3686

3687

3688

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

3689

3690

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3691

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

3692

3693

where the difference operator refers to set difference.

3694 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$3695 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3696 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 3697 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$3698 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$3699 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$3700 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$3701 \quad$$

$$3702 \quad$$

3703 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

3704 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 3705 using what is called a *standard vector mask and replace*. This is carried out under control of the
 3706 mask which acts as a “write mask”.

3707 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 3708 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$3709 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3710 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 3711 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 3712 mask are unchanged:

$$3713 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3714 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 3715 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 3716 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 3717 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

3718 4.3.7.2 `assign`: Standard matrix variant

3719 Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices.
 3720 The dimensions of the input matrix are the same size as the row and column index arrays provided.

3721 C Syntax

```
3722     GrB_Info GrB_assign(GrB_Matrix      C,
3723                       const GrB_Matrix Mask,
3724                       const GrB_BinaryOp accum,
3725                       const GrB_Matrix  A,
3726                       const GrB_Index   *row_indices,
3727                       GrB_Index         nrows,
3728                       const GrB_Index   *col_indices,
3729                       GrB_Index         ncols,
3730                       const GrB_Descriptor desc);
```

3731 **Parameters**

3732 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
3733 that may be accumulated with the result of the assign operation. On output, the
3734 matrix holds the results of the operation.

3735 Mask (IN) An optional “write” mask that controls which results from this operation are
3736 stored into the output matrix C. The mask dimensions must match those of the
3737 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
3738 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
3739 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
3740 dimensions of C), GrB_NULL should be specified.

3741 accum (IN) An optional binary operator used for accumulating entries into existing C
3742 entries. If assignment rather than accumulation is desired, GrB_NULL should be
3743 specified.

3744 A (IN) The GraphBLAS matrix whose contents are assigned to a subset of C.

3745 row_indices (IN) Pointer to the ordered set (array) of indices corresponding to the rows of C
3746 that are assigned. If all rows of C are to be assigned in order from 0 to `nrows - 1`,
3747 then GrB_ALL can be specified. Regardless of execution mode and return value,
3748 this array may be manipulated by the caller after this operation returns without
3749 affecting any deferred computations for this operation. If this array contains du-
3750 plicate values, it implies assignment of more than one value to the same location
3751 which leads to undefined results.

3752 nrows (IN) The number of values in the row_indices array. Must be equal to `nrows(A)` if
3753 A is not tranposed, or equal to `ncols(A)` if A is tranposed.

3754 col_indices (IN) Pointer to the ordered set (array) of indices corresponding to the columns
3755 of C that are assigned. If all columns of C are to be assigned in order from 0 to
3756 `ncols - 1`, then GrB_ALL should be specified. Regardless of execution mode and
3757 return value, this array may be manipulated by the caller after this operation
3758 returns without affecting any deferred computations for this operation. If this
3759 array contains duplicate values, it implies assignment of more than one value to
3760 the same location which leads to undefined results.

3761 ncols (IN) The number of values in col_indices array. Must be equal to `ncols(A)` if A is
3762 not tranposed, or equal to `nrows(A)` if A is tranposed.

3763 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
3764 should be specified. Non-default field/value pairs are listed as follows:
3765

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

3766

3767 Return Values

3768

3769

3770

3771

3772

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.

3773

GrB_PANIC Unknown internal error.

3774

3775

3776

3777

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.

3778

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

3779

3780

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).

3781

3782

3783

GrB_INDEX_OUT_OF_BOUNDS A value in `row_indices` is greater than or equal to `nrows(C)`, or a value in `col_indices` is greater than or equal to `ncols(C)`. In non-blocking mode, this can be reported as an execution error.

3784

3785

GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, `nrows` \neq `nrows(A)`, or `ncols` \neq `ncols(A)`.

3786

3787

3788

3789

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` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3790

3791

GrB_NULL_POINTER Either argument `row_indices` is a NULL pointer, argument `col_indices` is a NULL pointer, or both.

3792 **Description**

3793 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows
 3794 and columns in `C` in a specified order: $C(\text{row_indices}, \text{col_indices}) = A$; or, if an optional binary
 3795 accumulation operator (\odot) is provided, $C(\text{row_indices}, \text{col_indices}) = C(\text{row_indices}, \text{col_indices}) \odot A$.
 3796 More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned}
 & C(\text{row_indices}[i], \text{col_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 3797 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot A(i, j), \\
 & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

3798 Logically, this operation occurs in three steps:

3799 Setup The internal matrices and mask used in the computation are formed and their domains
 3800 and dimensions are tested for compatibility.

3801 Compute The indicated computations are carried out.

3802 Output The result is written into the output matrix, possibly under control of a mask.

3803 Up to three argument matrices are used in the `GrB_assign` operation:

- 3804 1. `C` = $\langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3805 2. `Mask` = $\langle \mathbf{D}(\text{Mask}), \text{nrows}(\text{Mask}), \text{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$ (optional)
- 3806 3. `A` = $\langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

3807 The argument matrices and the accumulation operator (if provided) are tested for domain compat-
 3808 ibility as follows:

- 3809 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 3810 must be from one of the pre-defined types of Table 2.2.
- 3811 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$.
- 3812 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 3813 of the accumulation operator and $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 3814 mulation operator.

3815 Two domains are compatible with each other if values from one domain can be cast to values in
 3816 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 3817 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 3818 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 3819 error listed above is returned.

3820 From the arguments, the internal matrices, mask, and index arrays used in the computation are
 3821 formed (\leftarrow denotes copy):

- 3822 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 3823 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
- 3824 (a) If `Mask = GrB.NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
3825 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 3826 (b) If `Mask \neq GrB.NULL`,
- 3827 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
3828 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
- 3829 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
3830 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
- 3831 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 3832 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 3833 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
- 3834 (a) If `row_indices = GrB.ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
3835 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 3836 5. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
- 3837 (a) If `col_indices = GrB.ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
3838 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

3839 The internal matrices and mask are checked for dimension compatibility. The following conditions
3840 must hold:

- 3841 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 3842 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 3843 3. $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$.
- 3844 4. $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$.

3845 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
3846 error listed above is returned.

3847 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
3848 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3849 We are now ready to carry out the assign and any additional associated operations. We describe
3850 this in terms of two intermediate vectors:

- 3851 • $\tilde{\mathbf{T}}$: The matrix holding the contents from $\tilde{\mathbf{A}}$ in their destination locations relative to $\tilde{\mathbf{C}}$.
- 3852 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

3853 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$3854 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

3855 At this point, if any value in the $\tilde{\mathbf{I}}$ array is not in the range $[0, \mathbf{nrows}(\tilde{\mathbf{C}}))$ or any value in the
 3856 $\tilde{\mathbf{J}}$ array is not in the range $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-
 3857 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred
 3858 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix \mathbf{C} is invalid from
 3859 this point forward in the sequence.

3860 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 3861 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$3862 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 3863 \quad \{(i, j, Z_{ij}) \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3864 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 3865 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
 3866 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

3867 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3868 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3869 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3870 \\ 3871 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

3872 where the difference operator refers to set difference.

- 3873 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$3874 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3875 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 3876 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$3877 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3878 \\ 3879 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3880 \\ 3881 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3882 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

3883 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 3884 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 3885 mask which acts as a “write mask”.

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

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

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

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

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

4.3.7.3 assign: Column variant

Assign the contents a vector to a subset of elements in one column of a matrix. Note that since the output cannot be transposed, a different variant of assign is provided to assign to a row of a matrix.

C Syntax

```

GrB_Info GrB_assign(GrB_Matrix      C,
                   const GrB_Vector  mask,
                   const GrB_BinaryOp accum,
                   const GrB_Vector  u,
                   const GrB_Index   *row_indices,
                   GrB_Index         nrows,
                   GrB_Index         col_index,
                   const GrB_Descriptor desc);

```

Parameters

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

mask (IN) An optional “write” mask that controls which results from this operation are stored into the specified column of the output matrix C. The mask dimensions must match those of a single column of the matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain of the Mask matrix must be of type

3919 bool or any of the predefined “built-in” types in Table 2.2. If the default mask
 3920 is desired (i.e., a mask that is all true with the dimensions of a column of C),
 3921 GrB_NULL should be specified.

3922 **accum** (IN) An optional binary operator used for accumulating entries into existing C
 3923 entries. If assignment rather than accumulation is desired, GrB_NULL should be
 3924 specified.

3925 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column
 3926 of C.

3927 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
 3928 the specified column of C that are to be assigned. If all elements of the column
 3929 in C are to be assigned in order from index 0 to `nrows - 1`, then GrB_ALL should
 3930 be specified. Regardless of execution mode and return value, this array may be
 3931 manipulated by the caller after this operation returns without affecting any de-
 3932 ferred computations for this operation. If this array contains duplicate values, it
 3933 implies in assignment of more than one value to the same location which leads to
 3934 undefined results.

3935 **nrows** (IN) The number of values in `row_indices` array. Must be equal to `size(u)`.

3936 **col_index** (IN) The index of the column in C to assign. Must be in the range `[0, ncols(C))`.

3937 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
 3938 should be specified. Non-default field/value pairs are listed as follows:
 3939

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output column in C is cleared (all elements removed) before result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <code>mask</code> .

3941 Return Values

3942 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 3943 blocking mode, this indicates that the compatibility tests on di-
 3944 mensions and domains for the input arguments passed successfully.
 3945 Either way, output matrix C is ready to be used in the next method
 3946 of the sequence.

3947 GrB_PANIC Unknown internal error.

3948 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
3949 GraphBLAS objects (input or output) is in an invalid state caused
3950 by a previous execution error. Call `GrB_error()` to access any error
3951 messages generated by the implementation.

3952 GrB_OUT_OF_MEMORY Not enough memory available for operation.

3953 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
3954 a call to `new` (or `dup` for vector or matrix parameters).

3955 GrB_INVALID_INDEX `col_index` is outside the allowable range (i.e., greater than `ncols(C)`).

3956 GrB_INDEX_OUT_OF_BOUNDS A value in `row_indices` is greater than or equal to `nrows(C)`. In
3957 non-blocking mode, this can be reported as an execution error.

3958 GrB_DIMENSION_MISMATCH `mask` size and number of rows in `C` are not the same, or `nrows` \neq
3959 `size(u)`.

3960 GrB_DOMAIN_MISMATCH The domains of the matrix and vector are incompatible with each
3961 other or the corresponding domains of the accumulation operator,
3962 or the mask's domain is not compatible with `bool` (in the case where
3963 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

3964 GrB_NULL_POINTER Argument `row_indices` is a NULL pointer.

3965 Description

3966 This variant of `GrB_assign` computes the result of assigning a subset of locations in a column of a
3967 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:
3968 $C(:, col_index) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(:, col_index) =$
3969 $C(:, col_index) \odot u$. Taking order of `row_indices` into account, it is more explicitly written as:

$$3970 \quad C(\text{row_indices}[i], \text{col_index}) = u(i), \quad \forall i : 0 \leq i < \text{nrows}, \text{ or}$$

$$3970 \quad C(\text{row_indices}[i], \text{col_index}) = C(\text{row_indices}[i], \text{col_index}) \odot u(i), \quad \forall i : 0 \leq i < \text{nrows}.$$

3971 Logically, this operation occurs in three steps:

3972 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
3973 domains and dimensions are tested for compatibility.

3974 **Compute** The indicated computations are carried out.

3975 **Output** The result is written into the output matrix, possibly under control of a mask.

3976 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 3977 1. $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3978 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

3979 3. $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

3980 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
3981 compatibility as follows:

3982 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
3983 must be from one of the pre-defined types of Table 2.2.

3984 2. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}(\mathbf{u})$.

3985 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
3986 of the accumulation operator and $\mathbf{D}(\mathbf{u})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
3987 mulation operator.

3988 Two domains are compatible with each other if values from one domain can be cast to values in
3989 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
3990 compatible with each other. A domain from a user-defined type is only compatible with itself. If
3991 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
3992 error listed above is returned.

3993 The `col_index` parameter is checked for a valid value. The following condition must hold:

3994 1. $0 \leq \text{col_index} < \mathbf{ncols}(\mathbf{C})$

3995 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
3996 is returned.

3997 From the arguments, the internal vectors, `mask`, and index array used in the computation are
3998 formed (\leftarrow denotes copy):

3999 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a column of \mathbf{C} as follows:

4000
$$\tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4001 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

4002 (a) If `mask` = `GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$.

4003 (b) If `mask` \neq `GrB_NULL`,

4004 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

4005 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

4006 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

4007 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4008 4. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:

4009 (a) If `row_indices` = `GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.

4010 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \text{row_indices}[i]$, $\forall i : 0 \leq i < \text{nrows}$.

4011 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
 4012 conditions must hold:

4013 1. $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$

4014 2. $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$.

4015 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
 4016 error listed above is returned.

4017 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4018 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4019 We are now ready to carry out the assign and any additional associated operations. We describe
 4020 this in terms of two intermediate vectors:

- 4021 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.
- 4022 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4023 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4024 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4025 At this point, if any value of $\tilde{\mathbf{I}}[i]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
 4026 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
 4027 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
 4028 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4029 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4030 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4031 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4032 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 4033 of $\tilde{\mathbf{c}}$ ($\text{ind}(\tilde{\mathbf{c}})$) and remove from it all the indices of $\tilde{\mathbf{c}}$ that are in the set of indices being
 4034 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\text{ind}(\tilde{\mathbf{t}})$).

4035 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4036 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$4037 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4038 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4040 where the difference operator refers to set difference.

4041 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4042 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4043 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4044 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4045 \quad z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$4046 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$4047 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

4048 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4049 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
4050 result matrix, $\mathbf{C}(:, \text{col_index})$. This is carried out under control of the mask which acts as a “write
4051 mask”.

4052 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(:, \text{col_index})$ on input to this
4053 operation are deleted and the new contents of the column is given by:

$$4054 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4055 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4056 copied into the column of the final result matrix, $\mathbf{C}(:, \text{col_index})$, and elements of this column
4057 that fall outside the set indicated by the mask are unchanged:

$$4058 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col_index}\} \cup$$

$$4059 \quad \{(i, \text{col_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup$$

$$4060 \quad \{(i, \text{col_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4061 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
4062 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4063 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may
4064 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

4065 4.3.7.4 `assign`: Row variant

4066 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the
4067 output cannot be transposed, a different variant of `assign` is provided to assign to a column of a
4068 matrix.

4071 C Syntax

```
4072         GrB_Info GrB_assign(GrB_Matrix      C,  
4073                             const GrB_Vector  mask,  
4074                             const GrB_BinaryOp accum,  
4075                             const GrB_Vector  u,  
4076                             GrB_Index        row_index,  
4077                             const GrB_Index   *col_indices,  
4078                             GrB_Index        ncols,  
4079                             const GrB_Descriptor desc);
```

4080 Parameters

4081 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values
4082 that may be accumulated with the result of the assign operation. On output, this
4083 matrix holds the results of the operation.

4084 **mask** (IN) An optional “write” mask that controls which results from this operation are
4085 stored into the specified row of the output matrix **C**. The mask dimensions must
4086 match those of a single row of the matrix **C**. If the **GrB_STRUCTURE** descriptor is
4087 *not* set for the mask, the domain of the Mask matrix must be of type **bool** or any
4088 of the predefined “built-in” types in Table 2.2. If the default mask is desired (i.e.,
4089 a mask that is all true with the dimensions of a row of **C**), **GrB_NULL** should be
4090 specified.

4091 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**
4092 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
4093 specified.

4094 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of
4095 **C**.

4096 **row_index** (IN) The index of the row in **C** to assign. Must be in the range $[0, \mathbf{nrows}(C))$.

4097 **col_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4098 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to
4099 be assigned in order from index 0 to $\mathbf{ncols} - 1$, then **GrB_ALL** should be specified.
4100 Regardless of execution mode and return value, this array may be manipulated by
4101 the caller after this operation returns without affecting any deferred computations
4102 for this operation. If this array contains duplicate values, it implies in assignment
4103 of more than one value to the same location which leads to undefined results.

4104 **ncols** (IN) The number of values in **col_indices** array. Must be equal to **size(u)**.

4105 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
4106 should be specified. Non-default field/value pairs are listed as follows:
4107

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <code>mask</code> .

4108

4109 Return Values

4110 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
4111 blocking mode, this indicates that the compatibility tests on di-
4112 mensions and domains for the input arguments passed successfully.
4113 Either way, output matrix C is ready to be used in the next method
4114 of the sequence.

4115 GrB_PANIC Unknown internal error.

4116 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
4117 GraphBLAS objects (input or output) is in an invalid state caused
4118 by a previous execution error. Call `GrB_error()` to access any error
4119 messages generated by the implementation.

4120 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4121 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
4122 a call to `new` (or `dup` for vector or matrix parameters).

4123 GrB_INVALID_INDEX `row_index` is outside the allowable range (i.e., greater than `nrows(C)`).

4124 GrB_INDEX_OUT_OF_BOUNDS A value in `col_indices` is greater than or equal to `ncols(C)`. In non-
4125 blocking mode, this can be reported as an execution error.

4126 GrB_DIMENSION_MISMATCH `mask` size and number of columns in C are not the same, or `ncols` \neq
4127 `size(u)`.

4128 GrB_DOMAIN_MISMATCH The domains of the matrix and vector are incompatible with each
4129 other or the corresponding domains of the accumulation operator,
4130 or the mask's domain is not compatible with `bool` (in the case where
4131 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4132 GrB_NULL_POINTER Argument `col_indices` is a NULL pointer.

4133 Description

4134 This variant of `GrB_assign` computes the result of assigning a subset of locations in a row of a
4135 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

4136 $C(\text{row_index}, :) = u$; or, if an optional binary accumulation operator (\odot) is provided, $C(\text{row_index}, :$
 4137 $) = C(\text{row_index}, :) \odot u$. Taking order of `col_indices` into account it is more explicitly written as:

$$4138 \quad C(\text{row_index}, \text{col_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$$

$$C(\text{row_index}, \text{col_indices}[j]) = C(\text{row_index}, \text{col_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$$

4139 Logically, this operation occurs in three steps:

4140 **Setup** The internal matrices, vectors and mask used in the computation are formed and their
 4141 domains and dimensions are tested for compatibility.

4142 **Compute** The indicated computations are carried out.

4143 **Output** The result is written into the output matrix, possibly under control of a mask.

4144 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 4145 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4146 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4147 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4148 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain
 4149 compatibility as follows:

- 4150 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 4151 must be from one of the pre-defined types of Table 2.2.
- 4152 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(u)$.
- 4153 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4154 of the accumulation operator and $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4155 mulation operator.

4156 Two domains are compatible with each other if values from one domain can be cast to values in
 4157 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 4158 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4159 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4160 error listed above is returned.

4161 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 4162 1. $0 \leq \text{row_index} < \mathbf{nrows}(C)$

4163 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above
 4164 is returned.

4165 From the arguments, the internal vectors, mask, and index array used in the computation are
 4166 formed (\leftarrow denotes copy):

4167 1. The vector, $\tilde{\mathbf{c}}$, is extracted from a row of \mathbf{C} as follows:

$$4168 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4169 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:

4170 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$.

4171 (b) If `mask \neq GrB_NULL`,

4172 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$,

4173 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$.

4174 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$.

4175 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4176 4. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:

4177 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.

4178 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \text{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4179 The internal vectors, matrices, and masks are checked for dimension compatibility. The following
4180 conditions must hold:

4181 1. $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

4182 2. $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$.

4183 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
4184 error listed above is returned.

4185 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4186 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4187 We are now ready to carry out the assign and any additional associated operations. We describe
4188 this in terms of two intermediate vectors:

4189 • $\tilde{\mathbf{t}}$: The vector holding the elements from $\tilde{\mathbf{u}}$ in their destination locations relative to $\tilde{\mathbf{c}}$.

4190 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4191 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4192 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4193 At this point, if any value of $\tilde{\mathbf{J}}[j]$ is outside the valid range of indices for vector $\tilde{\mathbf{c}}$, computation
4194 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`
4195 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the
4196 result matrix, \mathbf{C} , is invalid from this point forward in the sequence.

4197 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

4198 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4199 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4200 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 4201 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
 4202 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

4203 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4204 indices in $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{t}}$.

$$4205 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$4206 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

4207 where the difference operator refers to set difference.

4208 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4209 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4210 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4211 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4212 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$4213 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$4214 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

4215 where $\odot = \odot(\mathbf{accum})$, and the difference operator refers to set difference.

4216 Finally, the set of output values that make up the $\tilde{\mathbf{z}}$ vector are written into the column of the final
 4217 result matrix, $\mathbf{C}(\text{row_index}, :)$. This is carried out under control of the mask which acts as a “write
 4218 mask”.

4219 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in $\mathbf{C}(\text{row_index}, :)$ on input to this
 4220 operation are deleted and the new contents of the column is given by:

$$4221 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4222 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 4223 copied into the column of the final result matrix, $\mathbf{C}(\text{row_index}, :)$, and elements of this column
 4224 that fall outside the set indicated by the mask are unchanged:

$$4225 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row_index}\} \cup$$

$$4226 \quad \{(\text{row_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$4227 \quad \{(\text{row_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4228 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
 4229 of vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
 4230 exits with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may
 4231 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

4235 4.3.7.5 assign: Constant vector variant

4236 Assign the same value to a specified subset of vector elements. With the use of `GrB_ALL`, the entire
4237 destination vector can be filled with the constant.

4238 C Syntax

```
4239         GrB_Info GrB_assign(GrB_Vector          w,  
4240                           const GrB_Vector    mask,  
4241                           const GrB_BinaryOp   accum,  
4242                           <type>             val,  
4243                           const GrB_Index     *indices,  
4244                           GrB_Index          nindices,  
4245                           const GrB_Descriptor desc);
```

4246 Parameters

4247 `w` (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4248 that may be accumulated with the result of the assign operation. On output, this
4249 vector holds the results of the operation.

4250 `mask` (IN) An optional “write” mask that controls which results from this operation are
4251 stored into the output vector `w`. The mask dimensions must match those of the
4252 vector `w`. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
4253 of the mask vector must be of type `bool` or any of the predefined “built-in” types
4254 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
4255 dimensions of `w`), `GrB_NULL` should be specified.

4256 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`
4257 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
4258 specified.

4259 `val` (IN) Scalar value to assign to (a subset of) `w`.

4260 `indices` (IN) Pointer to the ordered set (array) of indices corresponding to the locations in
4261 `w` that are to be assigned. If all elements of `w` are to be assigned in order from 0
4262 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode
4263 and return value, this array may be manipulated by the caller after this operation
4264 returns without affecting any deferred computations for this operation. In this
4265 variant, the specific order of the values in the array has no effect on the result.
4266 Unlike other variants, if there are duplicated values in this array the result is still
4267 defined.

4268 `nindices` (IN) The number of values in `indices` array. Must be in the range: $[0, \text{size}(w)]$. If
4269 `nindices` is zero, the operation becomes a NO-OP.

4270
4271
4272

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.

4273

4274 Return Values

4275
4276
4277
4278
4279

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.

4280

GrB_PANIC Unknown internal error.

4281
4282
4283
4284

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.

4285

GrB_OUT_OF_MEMORY Not enough memory available for operation.

4286
4287

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

4288
4289

GrB_INDEX_OUT_OF_BOUNDS A value in indices is greater than or equal to size(w). In non-blocking mode, this can be reported as an execution error.

4290
4291

GrB_DIMENSION_MISMATCH mask and w dimensions are incompatible, or nindices is not less than size(w).

4292
4293
4294
4295

GrB_DOMAIN_MISMATCH The domains of the vector and scalar are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4296

GrB_NULL_POINTER Argument indices is a NULL pointer.

4297 **Description**

4298 This variant of GrB_assign computes the result of assigning a constant scalar value to locations in
4299 a destination GraphBLAS vector: $w(\text{indices}) = \text{val}$; or, if an optional binary accumulation operator
4300 (\odot) is provided, $w(\text{indices}) = w(\text{indices}) \odot \text{val}$. More explicitly:

$$\begin{aligned} 4301 \quad & w(\text{indices}[i]) = \text{val}, \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ & w(\text{indices}[i]) = w(\text{indices}[i]) \odot \text{val}, \forall i : 0 \leq i < \text{nindices}. \end{aligned}$$

4302 Logically, this operation occurs in three steps:

4303 **Setup** The internal vectors and mask used in the computation are formed and their domains
4304 and dimensions are tested for compatibility.

4305 **Compute** The indicated computations are carried out.

4306 **Output** The result is written into the output vector, possibly under control of a mask.

4307 Up to two argument vectors are used in the GrB_assign operation:

- 4308 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4309 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

4310 The argument scalar, vectors, and the accumulation operator (if provided) are tested for domain
4311 compatibility as follows:

- 4312 1. If mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\text{mask})$
4313 must be from one of the pre-defined types of Table 2.2.
- 4314 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}(\text{val})$.
- 4315 3. If accum is not GrB_NULL, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4316 of the accumulation operator and $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
4317 mulation operator.

4318 Two domains are compatible with each other if values from one domain can be cast to values in
4319 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
4320 compatible with each other. A domain from a user-defined type is only compatible with itself. If
4321 any compatibility rule above is violated, execution of GrB_assign ends and the domain mismatch
4322 error listed above is returned.

4323 From the arguments, the internal vectors, mask and index array used in the computation are formed
4324 (\leftarrow denotes copy):

- 4325 1. Vector $\tilde{w} \leftarrow w$.
- 4326 2. One-dimensional mask, \tilde{m} , is computed from argument mask as follows:

- 4327 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
- 4328 (b) If `mask \neq GrB_NULL`,
- 4329 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
- 4330 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
- 4331 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 4332 3. The internal index array, $\tilde{\mathbf{I}}$, is computed from argument indices as follows:
- 4333 (a) If `indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$.
- 4334 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$.

4335 The internal vector and mask are checked for dimension compatibility. The following conditions
4336 must hold:

- 4337 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 4338 2. $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$.

4339 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
4340 error listed above is returned.

4341 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4342 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4343 We are now ready to carry out the assign and any additional associated operations. We describe
4344 this in terms of two intermediate vectors:

- 4345 • $\tilde{\mathbf{t}}$: The vector holding the copies of the scalar `val` in their destination locations relative to $\tilde{\mathbf{w}}$.
- 4346 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4347 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4348 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

4349 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
4350 is not in the range $[0, \mathbf{size}(\tilde{\mathbf{w}}))$, the execution of `GrB_assign` ends and the index out-of-bounds
4351 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a
4352 sequence-terminating `GrB_wait()` is called. Regardless, the result vector, \mathbf{w} , is invalid from this
4353 point forward in the sequence.

4354 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows:

- 4355 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}}$ is defined as

$$4356 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4357 The above expression defines the structure of vector $\tilde{\mathbf{z}}$ as follows: We start with the structure
 4358 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
 4359 assigned ($\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$). Finally, we add the structure of $\tilde{\mathbf{t}}$ ($\mathbf{ind}(\tilde{\mathbf{t}})$).

4360 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4361 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4362 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$4363 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

4365 where the difference operator refers to set difference. We note that in this case of assigning
 4366 a constant, $\{\tilde{\mathbf{I}}[k], \forall k\}$ and $\mathbf{ind}(\tilde{\mathbf{t}})$ are identical.

- 4367 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4368 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4369 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
 4370 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$4371 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$4372 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$4373 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

4376 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4377 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
 4378 using what is called a *standard vector mask and replace*. This is carried out under control of the
 4379 mask which acts as a “write mask”.

- 4380 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
 4381 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4382 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 4383 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
 4384 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
 4385 mask are unchanged:

$$4386 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4387 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
 4388 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
 4389 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
 4390 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4391 **4.3.7.6 assign: Constant matrix variant**

4392 Assign the same value to a specified subset of matrix elements. With the use of `GrB_ALL`, the entire
4393 destination matrix can be filled with the constant.

4394 **C Syntax**

```
4395         GrB_Info GrB_assign(GrB_Matrix      C,  
4396                           const GrB_Matrix  Mask,  
4397                           const GrB_BinaryOp accum,  
4398                           <type>          val,  
4399                           const GrB_Index   *row_indices,  
4400                           GrB_Index       nrows,  
4401                           const GrB_Index   *col_indices,  
4402                           GrB_Index       ncols,  
4403                           const GrB_Descriptor desc);
```

4404 **Parameters**

4405 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
4406 that may be accumulated with the result of the assign operation. On output, the
4407 matrix holds the results of the operation.

4408 **Mask** (IN) An optional “write” mask that controls which results from this operation are
4409 stored into the output matrix `C`. The mask dimensions must match those of the
4410 matrix `C`. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
4411 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types
4412 in Table 2.2. If the default mask is desired (i.e., a mask that is all `true` with the
4413 dimensions of `C`), `GrB_NULL` should be specified.

4414 **accum** (IN) An optional binary operator used for accumulating entries into existing `C`
4415 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
4416 specified.

4417 **val** (IN) Scalar value to assign to (a subset of) `C`.

4418 **row_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of `C`
4419 that are assigned. If all rows of `C` are to be assigned in order from 0 to `nrows - 1`,
4420 then `GrB_ALL` can be specified. Regardless of execution mode and return value,
4421 this array may be manipulated by the caller after this operation returns without
4422 affecting any deferred computations for this operation. Unlike other variants, if
4423 there are duplicated values in this array the result is still defined.

4424 **nrows** (IN) The number of values in `row_indices` array. Must be in the range: `[0, nrows(C)]`.
4425 If `nrows` is zero, the operation becomes a NO-OP.

4426 `col_indices` (IN) Pointer to the ordered set (array) of indices corresponding to the columns of `C`
 4427 that are assigned. If all columns of `C` are to be assigned in order from 0 to `ncols-1`,
 4428 then `GrB_ALL` should be specified. Regardless of execution mode and return value,
 4429 this array may be manipulated by the caller after this operation returns without
 4430 affecting any deferred computations for this operation. Unlike other variants, if
 4431 there are duplicated values in this array the result is still defined.

4432 `ncols` (IN) The number of values in `col_indices` array. Must be in the range: `[0, ncols(C)]`.
 4433 If `ncols` is zero, the operation becomes a NO-OP.

4434 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 4435 should be specified. Non-default field/value pairs are listed as follows:
 4436

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix <code>C</code> is cleared (all elements removed) before the result is stored in it.
Mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <code>Mask</code> matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of <code>Mask</code> .

4438 Return Values

4439 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-
 4440 blocking mode, this indicates that the compatibility tests on di-
 4441 mensions and domains for the input arguments passed successfully.
 4442 Either way, output matrix `C` is ready to be used in the next method
 4443 of the sequence.

4444 `GrB_PANIC` Unknown internal error.

4445 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque
 4446 GraphBLAS objects (input or output) is in an invalid state caused
 4447 by a previous execution error. Call `GrB_error()` to access any error
 4448 messages generated by the implementation.

4449 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

4450 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by
 4451 a call to `new` (or `dup` for vector parameters).

4452 `GrB_INDEX_OUT_OF_BOUNDS` A value in `row_indices` is greater than or equal to `nrows(C)`, or a
 4453 value in `col_indices` is greater than or equal to `ncols(C)`. In non-
 4454 blocking mode, this can be reported as an execution error.

4455 `GrB_DIMENSION_MISMATCH` `Mask` and `C` dimensions are incompatible, `nrows` is not less than
 4456 `nrows(C)`, or `ncols` is not less than `ncols(C)`.

4457 GrB_DOMAIN_MISMATCH The domains of the matrix and scalar are incompatible with each
 4458 other or the corresponding domains of the accumulation operator,
 4459 or the mask's domain is not compatible with `bool` (in the case where
 4460 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4461 GrB_NULL_POINTER Either argument `row_indices` is a NULL pointer, argument `col_indices`
 4462 is a NULL pointer, or both.

4463 Description

4464 This variant of `GrB_assign` computes the result of assigning a constant scalar value to locations
 4465 in a destination GraphBLAS matrix: $C(\text{row_indices}, \text{col_indices}) = \text{val}$; or, if an optional binary
 4466 accumulation operator (\odot) is provided, $C(\text{row_indices}, \text{col_indices}) = w(\text{row_indices}, \text{col_indices}) \odot \text{val}$.
 4467 More explicitly:

$$\begin{aligned}
 & C(\text{row_indices}[i], \text{col_indices}[j]) = \text{val}, \text{ or} \\
 4468 & C(\text{row_indices}[i], \text{col_indices}[j]) = C(\text{row_indices}[i], \text{col_indices}[j]) \odot \text{val} \\
 & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4469 Logically, this operation occurs in three steps:

4470 Setup The internal vectors and mask used in the computation are formed and their domains
 4471 and dimensions are tested for compatibility.

4472 Compute The indicated computations are carried out.

4473 Output The result is written into the output matrix, possibly under control of a mask.

4474 Up to two argument matrices are used in the `GrB_assign` operation:

- 4475 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4476 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)

4477 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain
 4478 compatibility as follows:

- 4479 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 4480 must be from one of the pre-defined types of Table 2.2.
- 4481 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(\text{val})$.
- 4482 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 4483 of the accumulation operator and $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 4484 mulation operator.

4485 Two domains are compatible with each other if values from one domain can be cast to values in
 4486 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 4487 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 4488 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch
 4489 error listed above is returned.

4490 From the arguments, the internal matrices, index arrays, and mask used in the computation are
 4491 formed (\leftarrow denotes copy):

- 4492 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4493 2. Two-dimensional mask $\tilde{\mathbf{M}}$ is computed from argument `Mask` as follows:
 - 4494 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 4495 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 4496 (b) If `Mask \neq GrB_NULL`,
 - 4497 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 4498 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 4499 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 4500 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 4501 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4502 3. The internal row index array, $\tilde{\mathbf{I}}$, is computed from argument `row_indices` as follows:
 - 4503 (a) If `row_indices = GrB_ALL`, then $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$.
 - 4504 (b) Otherwise, $\tilde{\mathbf{I}}[i] = \mathbf{row_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$.
- 4505 4. The internal column index array, $\tilde{\mathbf{J}}$, is computed from argument `col_indices` as follows:
 - 4506 (a) If `col_indices = GrB_ALL`, then $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$.
 - 4507 (b) Otherwise, $\tilde{\mathbf{J}}[j] = \mathbf{col_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$.

4508 The internal matrix and mask are checked for dimension compatibility. The following conditions
 4509 must hold:

- 4510 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 4511 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 4512 3. $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$.
- 4513 4. $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$.

4514 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch
 4515 error listed above is returned.

4516 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4517 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4518 We are now ready to carry out the assign and any additional associated operations. We describe
 4519 this in terms of two intermediate vectors:

- 4520 • $\tilde{\mathbf{T}}$: The matrix holding the copies of the scalar `val` in their destination locations relative to
 4521 $\tilde{\mathbf{C}}$.
- 4522 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4523 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4524 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

4525 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
 4526 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
 4527 $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$, the execution of `GrB_assign` ends and the index out-of-bounds error listed above is
 4528 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating
 4529 `GrB_wait()` is called. Regardless, the result matrix `C` is invalid from this point forward in the
 4530 sequence.

4531 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows:

- 4532 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}}$ is defined as

$$4533 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\text{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4534 \quad \{(i, j, Z_{ij}) \mid (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4535 The above expression defines the structure of matrix $\tilde{\mathbf{Z}}$ as follows: We start with the structure
 4536 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
 4537 assigned ($\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$). Finally, we add the structure of $\tilde{\mathbf{T}}$ ($\mathbf{ind}(\tilde{\mathbf{T}})$).

4538 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4539 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4540 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4541 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4543 where the difference operator refers to set difference. We note that, in this particular case of
 4544 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.

- 4545 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4546 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4547 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 4548 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4549 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

4550

4551

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

4552

4553

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

4554

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

4555

4556

4557

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”.

4558

4559

- 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,

4560

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

4561

4562

4563

- 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:

4564

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

4565

4566

4567

4568

4569

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.

4570

4.3.8 apply: Apply a function to the elements of an object

4571

4572

Computes the transformation of the values of the elements of a vector or a matrix using a unary function, or a binary function where one argument is bound to a scalar.

4573

4.3.8.1 apply: Vector variant

4574

Computes the transformation of the values of the elements of a vector using a unary function.

4575

C Syntax

4576

4577

4578

4579

4580

4581

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

4582 **Parameters**

4583 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
 4584 that may be accumulated with the result of the apply operation. On output, this
 4585 vector holds the results of the operation.

4586 **mask** (IN) An optional “write” mask that controls which results from this operation are
 4587 stored into the output vector **w**. The mask dimensions must match those of the
 4588 vector **w**. If the **GrB_STRUCTURE** descriptor is *not* set for the mask, the domain
 4589 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types
 4590 in Table 2.2. If the default mask is desired (i.e., a mask that is all **true** with the
 4591 dimensions of **w**), **GrB_NULL** should be specified.

4592 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
 4593 entries. If assignment rather than accumulation is desired, **GrB_NULL** should be
 4594 specified.

4595 **op** (IN) A unary operator applied to each element of input vector **u**.

4596 **u** (IN) The GraphBLAS vector to which the unary function is applied.

4597 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
 4598 should be specified. Non-default field/value pairs are listed as follows:
 4599

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask .

4601 **Return Values**

4602 **GrB_SUCCESS** In blocking mode, the operation completed successfully. In non-
 4603 blocking mode, this indicates that the compatibility tests on di-
 4604 mensions and domains for the input arguments passed successfully.
 4605 Either way, output vector **w** is ready to be used in the next method
 4606 of the sequence.

4607 **GrB_PANIC** Unknown internal error.

4608 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
 4609 GraphBLAS objects (input or output) is in an invalid state caused
 4610 by a previous execution error. Call **GrB_error()** to access any error
 4611 messages generated by the implementation.

4612 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4613 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
4614 a call to `new` (or `dup` for vector parameters).

4615 GrB_DIMENSION_MISMATCH `mask`, `w` and/or `u` dimensions are incompatible.

4616 GrB_DOMAIN_MISMATCH The domains of the various vectors are incompatible with the corre-
4617 sponding domains of the accumulation operator or unary function,
4618 or the mask's domain is not compatible with `bool` (in the case where
4619 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4620 Description

4621 This variant of `GrB_apply` computes the result of applying a unary function to the elements of a
4622 GraphBLAS vector: $w = f(u)$; or, if an optional binary accumulation operator (\odot) is provided,
4623 $w = w \odot f(u)$.

4624 Logically, this operation occurs in three steps:

4625 **Setup** The internal vectors and mask used in the computation are formed and their domains
4626 and dimensions are tested for compatibility.

4627 **Compute** The indicated computations are carried out.

4628 **Output** The result is written into the output vector, possibly under control of a mask.

4629 Up to three argument vectors are used in this `GrB_apply` operation:

- 4630 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4631 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 4632 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4633 The argument vectors, unary operator and the accumulation operator (if provided) are tested for
4634 domain compatibility as follows:

- 4635 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4636 must be from one of the pre-defined types of Table 2.2.
- 4637 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.
- 4638 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4639 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
4640 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 4641 4. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in}(\text{op})$.

4642 Two domains are compatible with each other if values from one domain can be cast to values in
 4643 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
 4644 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 4645 If any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
 4646 error listed above is returned.

4647 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 4648 denotes copy):

- 4649 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 4650 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 4651 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 4652 (b) If `mask \neq GrB_NULL`,
 - 4653 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 4654 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 4655 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 4656 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4657 The internal vectors and masks are checked for dimension compatibility. The following conditions
 4658 must hold:

- 4659 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 4660 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

4661 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
 4662 error listed above is returned.

4663 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4664 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4665 We are now ready to carry out the apply and any additional associated operations. We describe
 4666 this in terms of two intermediate vectors:

- 4667 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the unary operator to the input vector $\tilde{\mathbf{u}}$.
- 4668 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4669 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$4670 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \mathbf{L}(\tilde{\mathbf{t}}) = \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

4671 where $f = \mathbf{f}(\mathbf{op})$.

4672 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

4673 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

4674 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

4675
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4676 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4677 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

4678
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

4679
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4680
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4681
4682

4683 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4684 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
4685 using what is called a *standard vector mask and replace*. This is carried out under control of the
4686 mask which acts as a “write mask”.

4687 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
4688 deleted and the content of the new output vector, \mathbf{w} , is defined as,

4689
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4690 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4691 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
4692 mask are unchanged:

4693
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\sim\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4694 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
4695 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
4696 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
4697 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4698 4.3.8.2 apply: Matrix variant

4699 Computes the transformation of the values of the elements of a matrix using a unary function.

4700 C Syntax

```
4701 GrB_Info GrB_apply(GrB_Matrix C,  
4702                   const GrB_Matrix Mask,  
4703                   const GrB_BinaryOp accum,  
4704                   const GrB_UnaryOp op,  
4705                   const GrB_Matrix A,  
4706                   const GrB_Descriptor desc);
```

4707 **Parameters**

4708 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 4709 that may be accumulated with the result of the apply operation. On output, the
 4710 matrix holds the results of the operation.

4711 Mask (IN) An optional “write” mask that controls which results from this operation are
 4712 stored into the output matrix C. The mask dimensions must match those of the
 4713 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 4714 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
 4715 in Table 2.2. If the default mask is desired (i.e., a mask that is all `true` with the
 4716 dimensions of C), `GrB_NULL` should be specified.

4717 accum (IN) An optional binary operator used for accumulating entries into existing C
 4718 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 4719 specified.

4720 op (IN) A unary operator applied to each element of input matrix A.

4721 A (IN) The GraphBLAS matrix to which the unary function is applied.

4722 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 4723 should be specified. Non-default field/value pairs are listed as follows:
 4724

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

4726 **Return Values**

4727 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 4728 blocking mode, this indicates that the compatibility tests on di-
 4729 mensions and domains for the input arguments passed successfully.
 4730 Either way, output matrix C is ready to be used in the next method
 4731 of the sequence.

4732 GrB_PANIC Unknown internal error.

4733 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 4734 GraphBLAS objects (input or output) is in an invalid state caused
 4735 by a previous execution error. Call `GrB_error()` to access any error
 4736 messages generated by the implementation.

4737 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

4738 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
4739 a call to `new` (or `Matrix_dup` for matrix parameters).

4740 GrB_INDEX_OUT_OF_BOUNDS A value in `row_indices` is greater than or equal to `nrows(A)`, or a
4741 value in `col_indices` is greater than or equal to `ncols(A)`. In non-
4742 blocking mode, this can be reported as an execution error.

4743 GrB_DIMENSION_MISMATCH Mask and C dimensions are incompatible, `nrows` \neq `nrows(C)`, or
4744 `ncols` \neq `ncols(C)`.

4745 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the cor-
4746 responding domains of the accumulation operator or unary func-
4747 tion, or the mask's domain is not compatible with `bool` (in the case
4748 where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4749 **Description**

4750 This variant of `GrB_apply` computes the result of applying a unary function to the elements of a
4751 GraphBLAS matrix: $C = f(A)$; or, if an optional binary accumulation operator (\odot) is provided,
4752 $C = C \odot f(A)$.

4753 Logically, this operation occurs in three steps:

4754 **Setup** The internal matrices and mask used in the computation are formed and their domains
4755 and dimensions are tested for compatibility.

4756 **Compute** The indicated computations are carried out.

4757 **Output** The result is written into the output matrix, possibly under control of a mask.

4758 Up to three argument matrices are used in the `GrB_apply` operation:

- 4759 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4760 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)
- 4761 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4762 The argument matrices, unary operator and the accumulation operator (if provided) are tested for
4763 domain compatibility as follows:

- 4764 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
4765 must be from one of the pre-defined types of Table 2.2.
- 4766 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the unary operator.

- 4767 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4768 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the unary operator must be compatible with
4769 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 4770 4. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in}(\text{op})$ of the unary operator.

4771 Two domains are compatible with each other if values from one domain can be cast to values in
4772 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
4773 all compatible with each other. A domain from a user-defined type is only compatible with itself.
4774 If any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
4775 error listed above is returned.

4776 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
4777 are formed (\leftarrow denotes copy):

- 4778 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 4779 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
- 4780 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
4781 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 4782 (b) If `Mask \neq GrB_NULL`,
- 4783 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
4784 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
- 4785 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
4786 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
- 4787 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 4788 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

4789 The internal matrices and mask are checked for dimension compatibility. The following conditions
4790 must hold:

- 4791 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 4792 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 4793 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 4794 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

4795 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
4796 error listed above is returned.

4797 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
4798 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4799 We are now ready to carry out the apply and any additional associated operations. We describe
4800 this in terms of two intermediate matrices:

- 4801 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the unary operator to the input matrix $\tilde{\mathbf{A}}$.
- 4802 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

4803 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as follows:

$$4804 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \mathbf{L}(\tilde{\mathbf{T}}) = \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\},$$

4805 where $f = \mathbf{f}(\text{op})$.

4806 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 4807 • If `accum = GrB.NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 4808 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$4809 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4810 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
4811 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$4812 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4813 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4814 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4815
4816
4817 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4818 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
4819 using what is called a *standard matrix mask and replace*. This is carried out under control of the
4820 mask which acts as a “write mask”.

- 4821 • If `desc[GrB_OUTP].GrB.REPLACE` is set, then any values in \mathbf{C} on input to this operation are
4822 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$4823 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4824 • If `desc[GrB_OUTP].GrB.REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
4825 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
4826 mask are unchanged:

$$4827 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4828 In `GrB_BLOCKING` mode, the method exits with return value `GrB.SUCCESS` and the new content
4829 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
4830 exits with return value `GrB.SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
4831 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
4832 sequence.

4833 4.3.8.3 apply: Vector-BinaryOp variants

4834 Computes the transformation of the values of the stored elements of a vector using a binary operator
4835 and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the first argument
4836 to the binary operator and stored elements of the vector are passed as the second argument. In the
4837 *bind-second* variant, the elements of the vector are passed as the first argument and the specified
4838 scalar value is passed as the second argument.

4839 C Syntax

```
4840 // bind-first
4841 GrB_Info GrB_apply(GrB_Vector          w,
4842                  const GrB_Vector      mask,
4843                  const GrB_BinaryOp    accum,
4844                  const GrB_BinaryOp    op,
4845                  <type>                val,
4846                  const GrB_Vector      u,
4847                  const GrB_Descriptor  desc);
4848
4849 // bind-second
4850 GrB_Info GrB_apply(GrB_Vector          w,
4851                  const GrB_Vector      mask,
4852                  const GrB_BinaryOp    accum,
4853                  const GrB_BinaryOp    op,
4854                  const GrB_Vector      u,
4855                  <type>                val,
4856                  const GrB_Descriptor  desc);
```

4857 Parameters

4858 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
4859 that may be accumulated with the result of the apply operation. On output, this
4860 vector holds the results of the operation.

4861 **mask** (IN) An optional “write” mask that controls which results from this operation are
4862 stored into the output vector **w**. The mask dimensions must match those of the
4863 vector **w**. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
4864 of the mask vector must be of type `bool` or any of the predefined “built-in” types
4865 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
4866 dimensions of **w**), `GrB_NULL` should be specified.

4867 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**
4868 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
4869 specified.

4870 op (IN) A binary operator applied to each element of input vector, *u*, and the scalar
4871 value, *val*.

4872 *u* (IN) The GraphBLAS vector whose elements are passed to the binary operator as
4873 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
4874 argument in the *bind-second* variant.

4875 *val* (IN) Scalar value that is passed to the binary operator as the left-hand (first)
4876 argument in the *bind-first* variant, or the right-hand (second) argument in the
4877 *bind-second* variant.

4878 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL
4879 should be specified. Non-default field/value pairs are listed as follows:
4880

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <i>w</i> is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <i>mask</i> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <i>mask</i> .

4882 Return Values

4883 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
4884 blocking mode, this indicates that the compatibility tests on di-
4885 mensions and domains for the input arguments passed successfully.
4886 Either way, output vector *w* is ready to be used in the next method
4887 of the sequence.

4888 GrB_PANIC Unknown internal error.

4889 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
4890 GraphBLAS objects (input or output) is in an invalid state caused
4891 by a previous execution error. Call GrB_error() to access any error
4892 messages generated by the implementation.

4893 GrB_OUT_OF_MEMORY Not enough memory available for operation.

4894 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
4895 a call to new (or dup for vector parameters).

4896 GrB_DIMENSION_MISMATCH *mask*, *w* and/or *u* dimensions are incompatible.

4897 GrB_DOMAIN_MISMATCH The domains of the various vectors and scalar are incompatible with
4898 the corresponding domains of the binary operator or accumulation
4899 operator, or the *mask*'s domain is not compatible with *bool* (in the
4900 case where desc[GrB_MASK].GrB_STRUCTURE is not set).

4901 **Description**

4902 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a
4903 GraphBLAS vector each composed with a scalar constant, `val`:

4904 bind-first: $w = f(\text{val}, u)$

4905 bind-second: $w = f(u, \text{val}),$

4906 or if an optional binary accumulation operator (\odot) is provided:

4907 bind-first: $w = w \odot f(\text{val}, u)$

4908 bind-second: $w = w \odot f(u, \text{val}).$

4909 Logically, this operation occurs in three steps:

4910 **Setup** The internal vectors and mask used in the computation are formed and their domains
4911 and dimensions are tested for compatibility.

4912 **Compute** The indicated computations are carried out.

4913 **Output** The result is written into the output vector, possibly under control of a mask.

4914 Up to three argument vectors are used in this `GrB_apply` operation:

4915 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

4916 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)

4917 3. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4918 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are
4919 tested for domain compatibility as follows:

4920 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
4921 must be from one of the pre-defined types of Table 2.2.

4922 2. $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.

4923 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
4924 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
4925 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

4926 4. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

4927 5. $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

4928 Two domains are compatible with each other if values from one domain can be cast to values in
 4929 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
 4930 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 4931 If any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
 4932 error listed above is returned.

4933 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 4934 denotes copy):

- 4935 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 4936 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 4937 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 4938 (b) If `mask \neq GrB_NULL`,
 - 4939 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 4940 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 4941 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 4942 3. Vector $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$.

4943 The internal vectors and masks are checked for dimension compatibility. The following conditions
 4944 must hold:

- 4945 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 4946 2. $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$.

4947 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
 4948 error listed above is returned.

4949 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 4950 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4951 We are now ready to carry out the apply and any additional associated operations. We describe
 4952 this in terms of two intermediate vectors:

- 4953 • $\tilde{\mathbf{t}}$: The vector holding the result from applying the binary operator to the input vector $\tilde{\mathbf{u}}$.
- 4954 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

4955 The intermediate vector, $\tilde{\mathbf{t}}$, is created as one of the following:

- 4956 bind-first: $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \mathbf{L}(\tilde{\mathbf{t}}) = \{(i, f(\mathbf{val}, \tilde{\mathbf{u}}(i))) \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle$,
- 4957 bind-second: $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \mathbf{L}(\tilde{\mathbf{t}}) = \{(i, f(\tilde{\mathbf{u}}(i), \mathbf{val})) \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle$,

4958 where $f = \mathbf{f}(\text{op})$.

4959 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

- 4960 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.
- 4961 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

$$4962 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4963 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
4964 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

$$\begin{aligned} 4965 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 4966 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 4967 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 4968 \end{aligned}$$

4970 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

4971 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
4972 using what is called a *standard vector mask and replace*. This is carried out under control of the
4973 mask which acts as a “write mask”.

- 4974 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
4975 deleted and the content of the new output vector, \mathbf{w} , is defined as,

$$4976 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 4977 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
4978 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
4979 mask are unchanged:

$$4980 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

4981 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
4982 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
4983 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
4984 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

4985 4.3.8.4 apply: Matrix-BinaryOp variants

4986 Computes the transformation of the values of the stored elements of a matrix using a binary
4987 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the
4988 first argument to the binary operator and stored elements of the matrix are passed as the second
4989 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument
4990 and the specified scalar value is passed as the second argument.

4991 C Syntax

```
4992 // bind-first
4993 GrB_Info GrB_apply(GrB_Matrix      C,
4994                   const GrB_Matrix Mask,
4995                   const GrB_BinaryOp accum,
4996                   const GrB_BinaryOp op,
4997                   <type>           val,
4998                   const GrB_Matrix  A,
4999                   const GrB_Descriptor desc);
5000
5001 // bind-second
5002 GrB_Info GrB_apply(GrB_Matrix      C,
5003                   const GrB_Matrix Mask,
5004                   const GrB_BinaryOp accum,
5005                   const GrB_BinaryOp op,
5006                   const GrB_Matrix  A,
5007                   <type>           val,
5008                   const GrB_Descriptor desc);
```

5009 Parameters

5010 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5011 that may be accumulated with the result of the apply operation. On output, the
5012 matrix holds the results of the operation.

5013 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5014 stored into the output matrix C. The mask dimensions must match those of the
5015 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5016 of the Mask matrix must be of type bool or any of the predefined “built-in” types
5017 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
5018 dimensions of C), GrB_NULL should be specified.

5019 **accum** (IN) An optional binary operator used for accumulating entries into existing C
5020 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5021 specified.

5022 **op** (IN) A binary operator applied to each element of input matrix, A, with the element
5023 of the input matrix used as the left-hand argument, and the scalar value, val, used
5024 as the right-hand argument.

5025 **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as
5026 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)
5027 argument in the *bind-second* variant.

5028 **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)
5029 argument in the *bind-first* variant, or the right-hand (second) argument in the

5030

bind-second variant.

5031

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:

5032

5033

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <code>Mask</code> matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of <code>Mask</code> .
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation (<i>bind-second</i> variant only).
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation (<i>bind-first</i> variant only).

5034

5035 Return Values

5036

`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.

5037

5038

5039

5040

5041

`GrB_PANIC` Unknown internal error.

5042

`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.

5043

5044

5045

5046

`GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

5047

`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).

5048

5049

`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.

5050

5051

5052

`GrB_DIMENSION_MISMATCH` `Mask` and `C` dimensions are incompatible, `nrows` \neq `nrows(C)`, or `ncols` \neq `ncols(C)`.

5053

5054

`GrB_DOMAIN_MISMATCH` The domains of the various matrices and scalar are incompatible with the corresponding domains of the binary operator or accumulation operator, or the mask's domain is not compatible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

5055

5056

5057

5058 **Description**

5059 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a
5060 GraphBLAS matrix each composed with a scalar constant, `val`:

5061 bind-first: $C = f(\text{val}, A)$

5062 bind-second: $C = f(A, \text{val});$

5063 or if an optional binary accumulation operator (\odot) is provided:

5064 bind-first: $C = C \odot f(\text{val}, A)$

5065 bind-second: $C = C \odot f(A, \text{val}).$

5066 Logically, this operation occurs in three steps:

5067 **Setup** The internal matrices and mask used in the computation are formed and their domains
5068 and dimensions are tested for compatibility.

5069 **Compute** The indicated computations are carried out.

5070 **Output** The result is written into the output matrix, possibly under control of a mask.

5071 Up to three argument matrices are used in the `GrB_apply` operation:

5072 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

5073 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)

5074 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5075 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are
5076 tested for domain compatibility as follows:

5077 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
5078 must be from one of the pre-defined types of Table 2.2.

5079 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{out}(\text{op})$ of the binary operator.

5080 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5081 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of the binary operator must be compatible with
5082 $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5083 4. $\mathbf{D}(A)$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$ of the binary operator.

5084 5. $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$ of the binary operator.

5085 Two domains are compatible with each other if values from one domain can be cast to values in
 5086 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are
 5087 all compatible with each other. A domain from a user-defined type is only compatible with itself.
 5088 If any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch
 5089 error listed above is returned.

5090 From the argument matrices, the internal matrices, mask, and index arrays used in the computation
 5091 are formed (\leftarrow denotes copy):

- 5092 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 5093 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 5094 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 5095 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 5096 (b) If `Mask \neq GrB_NULL`,
 - 5097 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 5098 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 5099 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 5100 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 5101 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 5102 3. Matrix $\tilde{\mathbf{A}}$ is computed from argument `A` as follows:

bind-first:	$\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP1}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$
bind-second:	$\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$

5105 The internal matrices and mask are checked for dimension compatibility. The following conditions
 5106 must hold:

- 5107 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 5108 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 5109 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.
- 5110 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.

5111 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch
 5112 error listed above is returned.

5113 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5114 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5115 We are now ready to carry out the apply and any additional associated operations. We describe
 5116 this in terms of two intermediate matrices:

- 5117 • $\tilde{\mathbf{T}}$: The matrix holding the result from applying the binary operator to the input matrix $\tilde{\mathbf{A}}$.

5118 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5119 The intermediate matrix, $\tilde{\mathbf{T}}$, is created as one of the following:

5120 bind-first: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \mathbf{L}(\tilde{\mathbf{T}}) = \{(i, j, f(\text{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

5121 bind-second: $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \mathbf{L}(\tilde{\mathbf{T}}) = \{(i, j, f(\tilde{\mathbf{A}}(i, j), \text{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

5122 where $f = \mathbf{f}(\text{op})$.

5123 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

5124 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.

5125 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

5126
$$\tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5127 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5128 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

5129
$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

5130
$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5131
$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5132 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5133 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5134 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5135 mask which acts as a “write mask”.

5136 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5137 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

5140
$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5141 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
5142 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
5143 mask are unchanged:

5144
$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5145 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5146 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5147 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
5148 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
5149 sequence.

5150 4.3.9 reduce: Perform a reduction across the elements of an object

5151 Computes the reduction of the values of the elements of a vector or matrix.

5152 4.3.9.1 reduce: Standard matrix to vector variant

5153 This performs a reduction across rows of a matrix to produce a vector. If column reduction
5154 across columns is desired, the input matrix should be transposed which can be specified using the
5155 descriptor.

5156 C Syntax

```
5157     GrB_Info GrB_reduce(GrB_Vector      w,  
5158                       const GrB_Vector mask,  
5159                       const GrB_BinaryOp accum,  
5160                       const GrB_Monoid op,  
5161                       const GrB_Matrix A,  
5162                       const GrB_Descriptor desc);  
5163  
5164     GrB_Info GrB_reduce(GrB_Vector      w,  
5165                       const GrB_Vector mask,  
5166                       const GrB_BinaryOp accum,  
5167                       const GrB_BinaryOp op,  
5168                       const GrB_Matrix A,  
5169                       const GrB_Descriptor desc);
```

5170 Parameters

5171 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values
5172 that may be accumulated with the result of the reduction operation. On output,
5173 this vector holds the results of the operation.

5174 **mask** (IN) An optional “write” mask that controls which results from this operation are
5175 stored into the output vector *w*. The mask dimensions must match those of the
5176 vector *w*. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain
5177 of the mask vector must be of type `bool` or any of the predefined “built-in” types
5178 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
5179 dimensions of *w*), `GrB_NULL` should be specified.

5180 **accum** (IN) An optional binary operator used for accumulating entries into existing *w*
5181 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
5182 specified.

5183 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.
5184 Depending on which type is passed, the following defines the binary operator with

5185 one domain, $F_b = \langle D, D, D, \oplus \rangle$, that is used:

5186 BinaryOp: $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$.

5187 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
5188 monoid is ignored.

5189 If `op` is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in
5190 both cases $\odot(\text{op})$ must be commutative and associative. Otherwise, the outcome
5191 of the operation is undefined.

5192 A (IN) The GraphBLAS matrix on which reduction will be performed.

5193 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
5194 should be specified. Non-default field/value pairs are listed as follows:
5195

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector <code>w</code> is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of <code>mask</code> .
A	GrB_INP0	GrB_TRAN	Use transpose of <code>A</code> for the operation.

5197 Return Values

5198 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
5199 blocking mode, this indicates that the compatibility tests on di-
5200 mensions and domains for the input arguments passed successfully.
5201 Either way, output vector `w` is ready to be used in the next method
5202 of the sequence.

5203 GrB_PANIC Unknown internal error.

5204 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
5205 GraphBLAS objects (input or output) is in an invalid state caused
5206 by a previous execution error. Call `GrB_error()` to access any error
5207 messages generated by the implementation.

5208 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

5209 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
5210 a call to `new` (or `dup` for vector parameters).

5211 GrB_DIMENSION_MISMATCH `mask`, `w` and/or `u` dimensions are incompatible.

5212 GrB_DOMAIN_MISMATCH Either the domains of the various vectors and matrices are incom-
 5213 compatible with the corresponding domains of the accumulation oper-
 5214 ator or reduce function, or the domains of the GraphBLAS binary
 5215 operator `op` are not all the same, or the mask's domain is not com-
 5216 patible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE`
 5217 is not set).

5218 **Description**

5219 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows
 5220 of an input matrix: $w(i) = \bigoplus A(i, :)\forall i$; or, if an optional binary accumulation operator is provided,
 5221 $w(i) = w(i) \odot (\bigoplus A(i, :))\forall i$, where $\bigoplus = \odot(F_b)$ and $\odot = \odot(\text{accum})$.

5222 Logically, this operation occurs in three steps:

5223 **Setup** The internal vector, matrix and mask used in the computation are formed and their
 5224 domains and dimensions are tested for compatibility.

5225 **Compute** The indicated computations are carried out.

5226 **Output** The result is written into the output vector, possibly under control of a mask.

5227 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 5228 1. $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5229 2. $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$ (optional)
- 5230 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5231 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested
 5232 for domain compatibility as follows:

- 5233 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{mask})$
 5234 must be from one of the pre-defined types of Table 2.2.
- 5235 2. $\mathbf{D}(w)$ must be compatible with the domain of the reduction binary operator, $\mathbf{D}(F_b)$.
- 5236 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(w)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5237 of the accumulation operator and $\mathbf{D}(F_b)$, must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of the accu-
 5238 mulation operator.
- 5239 4. $\mathbf{D}(A)$ must be compatible with the domain of the binary reduction operator, $\mathbf{D}(F_b)$.

5240 Two domains are compatible with each other if values from one domain can be cast to values in
 5241 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 5242 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5243 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 5244 error listed above is returned.

5245 From the argument vectors, the internal vectors and mask used in the computation are formed (\leftarrow
 5246 denotes copy):

- 5247 1. Vector $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$.
- 5248 2. One-dimensional mask, $\tilde{\mathbf{m}}$, is computed from argument `mask` as follows:
 - 5249 (a) If `mask = GrB_NULL`, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$.
 - 5250 (b) If `mask \neq GrB_NULL`,
 - 5251 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$,
 - 5252 ii. Otherwise, $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$.
 - 5253 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{m}} \leftarrow \neg\tilde{\mathbf{m}}$.
- 5254 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

5255 The internal vectors and masks are checked for dimension compatibility. The following conditions
 5256 must hold:

- 5257 1. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5258 2. $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

5259 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-
 5260 match error listed above is returned.

5261 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5262 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5263 We carry out the reduce and any additional associated operations. We describe this in terms of
 5264 two intermediate vectors:

- 5265 • $\tilde{\mathbf{t}}$: The vector holding the result from reducing along the rows of input matrix $\tilde{\mathbf{A}}$.
- 5266 • $\tilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

5267 The intermediate vector, $\tilde{\mathbf{t}}$, is created as follows:

$$5268 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \mathbf{L}(\tilde{\mathbf{t}}) = \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

5269 The value of each of its elements is computed by

$$5270 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

5271 where $\bigoplus = \bigodot(F_b)$.

5272 The intermediate vector $\tilde{\mathbf{z}}$ is created as follows, using what is called a *standard vector accumulate*:

5273 • If `accum = GrB_NULL`, then $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$.

5274 • If `accum` is a binary operator, then $\tilde{\mathbf{z}}$ is defined as

5275
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5276 The values of the elements of $\tilde{\mathbf{z}}$ are computed based on the relationships between the sets of
5277 indices in $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{t}}$.

5278
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

5279
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5280
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5281
5282

5283 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5284 Finally, the set of output values that make up vector $\tilde{\mathbf{z}}$ are written into the final result vector \mathbf{w} ,
5285 using what is called a *standard vector mask and replace*. This is carried out under control of the
5286 mask which acts as a “write mask”.

5287 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{w} on input to this operation are
5288 deleted and the content of the new output vector, \mathbf{w} , is defined as,

5289
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5290 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{z}}$ indicated by the mask are
5291 copied into the result vector, \mathbf{w} , and elements of \mathbf{w} that fall outside the set indicated by the
5292 mask are unchanged:

5293
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5294 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of
5295 vector \mathbf{w} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits
5296 with return value `GrB_SUCCESS` and the new content of vector \mathbf{w} is as defined above but may not
5297 be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

5298 4.3.9.2 reduce: Vector-scalar variant

5299 Reduce all stored values into a single scalar.

5300 C Syntax

```
5301      GrB_Info GrB_reduce(<type>          *val,  
5302                          const GrB_BinaryOp accum,  
5303                          const GrB_Monoid  op,  
5304                          const GrB_Vector  u,  
5305                          const GrB_Descriptor desc);
```

5306 **Parameters**

5307 **val** (INOUT) Scalar to store final reduced value into. On input, the scalar provides
5308 a value that may be accumulated with the result of the reduction operation. On
5309 output, this scalar holds the results of the operation.

5310 **accum** (IN) An optional binary operator used for accumulating entries into existing **val**
5311 value. If assignment rather than accumulation is desired, **GrB_NULL** should be
5312 specified.

5313 **op** (IN) The monoid used in the element-wise reduction operation, $M = \langle D, \oplus, 0 \rangle$.
5314 The binary operator, \oplus , must be commutative and associative; otherwise, the
5315 outcome of the operation is undefined.

5316 **u** (IN) The GraphBLAS vector on which reduction will be performed.

5317 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB_NULL**
5318 should be specified. Non-default field/value pairs are listed as follows:
5319

5320	Param	Field	Value	Description
------	-------	-------	-------	-------------

5321 *Note:* This argument is defined for consistency with the other GraphBLAS operations.
5322 There are currently no non-default field/value pairs that can be set for this
5323 operation.

5324 **Return Values**

5325 **GrB_SUCCESS** In blocking or non-blocking mode, the operation completed suc-
5326 cessfully, and the output scalar **val** is ready to be used in the next
5327 method of the sequence.

5328 **GrB_PANIC** Unknown internal error.

5329 **GrB_INVALID_OBJECT** This is returned in any execution mode whenever one of the opaque
5330 GraphBLAS objects (input or output) is in an invalid state caused
5331 by a previous execution error. Call **GrB_error()** to access any error
5332 messages generated by the implementation.

5333 **GrB_OUT_OF_MEMORY** Not enough memory available for the operation.

5334 **GrB_UNINITIALIZED_OBJECT** One or more of the GraphBLAS objects has not been initialized by
5335 a call to **new** (or **Vector_dup** for vector parameters).

5336 **GrB_DOMAIN_MISMATCH** The domains of input and output arguments are incompatible with
5337 the corresponding domains of the accumulation operator, or reduce
5338 operator.

5339 **GrB_NULL_POINTER** **val** pointer is **NULL**.

5340 **Description**

5341 This variant of `GrB_reduce` computes the result of performing a reduction across each of the elements
 5342 of an input vector: $\text{val} = \bigoplus u(\cdot)$; or, if an optional binary accumulation operator is provided,
 5343 $\text{val} = \text{val} \odot (\bigoplus u(\cdot))$, where $\bigoplus = \odot(\text{op})$ and $\odot = \odot(\text{accum})$.

5344 Logically, this operation occurs in three steps:

5345 **Setup** The internal vector used in the computation is formed and its domain is tested for
 5346 compatibility.

5347 **Compute** The indicated computations are carried out.

5348 **Output** The result is written into the output scalar.

5349 One vector argument is used in this `GrB_reduce` operation:

- 5350 1. $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5351 The output scalar, argument vector, reduction operator and accumulation operator (if provided)
 5352 are tested for domain compatibility as follows:

- 5353 1. If `accum` is `GrB_NULL`, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}(\text{op})$ of the reduction binary
 5354 operator.
- 5355 2. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5356 of the accumulation operator and $\mathbf{D}(\text{op})$ of the reduction binary operator must be compatible
 5357 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.
- 5358 3. $\mathbf{D}(u)$ must be compatible with $\mathbf{D}(\text{op})$ of the binary reduction operator.

5359 Two domains are compatible with each other if values from one domain can be cast to values in
 5360 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 5361 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5362 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 5363 error listed above is returned.

5364 From the argument vector, the internal vector used in the computation is formed (\leftarrow denotes copy):

- 5365 1. Vector $\tilde{\mathbf{u}} \leftarrow u$.

5366 We are now ready to carry out the reduce and any additional associated operations. First, an
 5367 intermediate scalar result t is computed using the recurrence:

5368
$$t = \begin{cases} \mathbf{0}(\text{op}), & \text{if } \mathbf{ind}(\tilde{\mathbf{u}}) = \emptyset, \\ \bigoplus_{i \in \mathbf{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i), & \text{otherwise.} \end{cases}$$

5369 Where $\oplus = \odot(\text{op})$, and $\mathbf{0}(\text{op})$ is the identity of the monoid.

5370 The final reduction value `val` is computed as follows:

- 5371 • If `accum = GrB_NULL`, then $\text{val} \leftarrow t$.
- 5372 • If `accum` is a binary operator, then $\text{val} \leftarrow \text{val} \odot t$, where $\odot = \odot(\text{accum})$.

5373 In both `GrB_BLOCKING` and `GrB_NONBLOCKING` modes, the method exits with return value
5374 `GrB_SUCCESS` and the new contents of `val` is as defined above.

5375 4.3.9.3 reduce: Matrix-scalar variant

5376 Reduce all stored values into a single scalar.

5377 C Syntax

```
5378     GrB_Info GrB_reduce(<type>          *val,  
5379                          const GrB_BinaryOp  accum,  
5380                          const GrB_Monoid    op,  
5381                          const GrB_Matrix    A,  
5382                          const GrB_Descriptor desc);
```

5383 Parameters

5384 `val` (INOUT) Scalar to store final reduced value into. On input, the scalar provides
5385 a value that may be accumulated with the result of the reduction operation. On
5386 output, this scalar holds the results of the operation.

5387 `accum` (IN) An optional binary operator used for accumulating entries into existing `val`
5388 value. If assignment rather than accumulation is desired, `GrB_NULL` should be
5389 specified.

5390 `op` (IN) The monoid used in the element-wise reduction operation, $M = \langle D, \oplus, 0 \rangle$.
5391 The binary operator, \oplus , must be commutative and associative; otherwise, the
5392 outcome of the operation is undefined.

5393 `A` (IN) The GraphBLAS matrix on which reduction will be performed.

5394 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
5395 should be specified. Non-default field/value pairs are listed as follows:

5397	Param	Field	Value	Description
------	-------	-------	-------	-------------

5398 *Note:* This argument is defined for consistency with the other GraphBLAS opera-
5399 tions. There are currently no non-default field/value pairs that can be set for this
5400 operation.

5401 **Return Values**

5402 GrB_SUCCESS In blocking or non-blocking mode, the operation completed suc-
5403 cessfully, and the output scalar `val` is ready to be used in the next
5404 method of the sequence.

5405 GrB_PANIC Unknown internal error.

5406 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
5407 GraphBLAS objects (input or output) is in an invalid state caused
5408 by a previous execution error. Call `GrB_error()` to access any error
5409 messages generated by the implementation.

5410 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

5411 GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by
5412 a call to `new` (or `Matrix_dup` for matrix parameters).

5413 GrB_DOMAIN_MISMATCH The domains of input and output arguments are incompatible with
5414 the corresponding domains of the accumulation operator, or reduce
5415 operator.

5416 GrB_NULL_POINTER `val` pointer is NULL.

5417 **Description**

5418 This variant of `GrB_reduce` computes the result of performing a reduction across each of the elements
5419 of an input matrix: $\text{val} = \bigoplus A(:, :)$; or, if an optional binary accumulation operator is provided,
5420 $\text{val} = \text{val} \odot (\bigoplus A(:, :))$, where $\bigoplus = \bigodot(\text{op})$ and $\odot = \bigodot(\text{accum})$.

5421 Logically, this operation occurs in three steps:

5422 **Setup** The internal matrix used in the computation is formed and its domain is tested for
5423 compatibility.

5424 **Compute** The indicated computations are carried out.

5425 **Output** The result is written into the output scalar.

5426 One matrix argument is used in this `GrB_reduce` operation:

5427 1. $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

5428 The output scalar, argument matrix, reduction operator and accumulation operator (if provided)
5429 are tested for domain compatibility as follows:

5430 1. If `accum` is `GrB_NULL`, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}(\text{op})$ of the reduction binary
5431 operator.

5432 2. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\text{val})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
 5433 of the accumulation operator and $\mathbf{D}(\text{op})$ of the reduction binary operator must be compatible
 5434 with $\mathbf{D}_{in_2}(\text{accum})$ of the accumulation operator.

5435 3. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}(\text{op})$ of the binary reduction operator.

5436 Two domains are compatible with each other if values from one domain can be cast to values in
 5437 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 5438 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5439 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch
 5440 error listed above is returned.

5441 From the argument matrix, the internal matrix used in the computation is formed (\leftarrow denotes
 5442 copy):

5443 1. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{A}$.

5444 We are now ready to carry out the reduce and any additional associated operations. First, an
 5445 intermediate scalar result t is computed using the recurrence:

$$5446 \quad t = \begin{cases} \mathbf{0}(\text{op}), & \text{if } \text{ind}(\tilde{\mathbf{A}}) = \emptyset, \\ \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j), & \text{otherwise.} \end{cases}$$

5447 Where $\oplus = \odot(\text{op})$, and $\mathbf{0}(\text{op})$ is the identity of the monoid.

5448 The final reduction value `val` is computed as follows:

- 5449 • If `accum = GrB_NULL`, then `val` $\leftarrow t$.
- 5450 • If `accum` is a binary operator, then `val` $\leftarrow \text{val} \odot t$, where $\odot = \odot(\text{accum})$.

5451 In both `GrB_BLOCKING` and `GrB_NONBLOCKING` modes, the method exits with return value
 5452 `GrB_SUCCESS` and the new contents of `val` is as defined above.

5453 4.3.10 transpose: Transpose rows and columns of a matrix

5454 This version computes a new matrix that is the transpose of the source matrix.

5455 C Syntax

```
5456     GrB_Info GrB_transpose(GrB_Matrix      C,
5457                          const GrB_Matrix Mask,
5458                          const GrB_BinaryOp accum,
5459                          const GrB_Matrix A,
5460                          const GrB_Descriptor desc);
```

5461 **Parameters**

5462 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
 5463 that may be accumulated with the result of the transpose operation. On output,
 5464 the matrix holds the results of the operation.

5465 Mask (IN) An optional “write” mask that controls which results from this operation are
 5466 stored into the output matrix C. The mask dimensions must match those of the
 5467 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
 5468 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types
 5469 in Table 2.2. If the default mask is desired (i.e., a mask that is all `true` with the
 5470 dimensions of C), `GrB_NULL` should be specified.

5471 accum (IN) An optional binary operator used for accumulating entries into existing C
 5472 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be
 5473 specified.

5474 A (IN) The GraphBLAS matrix on which transposition will be performed.

5475 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`
 5476 should be specified. Non-default field/value pairs are listed as follows:

5477

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_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for the operation.

5478

5479 **Return Values**

5480 GrB_SUCCESS In blocking mode, the operation completed successfully. In non-
 5481 blocking mode, this indicates that the compatibility tests on di-
 5482 mensions and domains for the input arguments passed successfully.
 5483 Either way, output matrix C is ready to be used in the next method
 5484 of the sequence.

5485 GrB_PANIC Unknown internal error.

5486 GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque
 5487 GraphBLAS objects (input or output) is in an invalid state caused
 5488 by a previous execution error. Call `GrB_error()` to access any error
 5489 messages generated by the implementation.

5490 GrB_OUT_OF_MEMORY Not enough memory available for the operation.

5491
5492
5493
5494
5495
5496
5497

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

GrB_DIMENSION_MISMATCH `mask`, `C` and/or `A` dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the corresponding domains of the accumulation operator, or the mask's domain is not compatible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

5498 Description

5499 **GrB_transpose** computes the result of performing a transpose of the input matrix: $C = A^T$; or, if an
5500 optional binary accumulation operator (\odot) is provided, $C = C \odot A^T$. We note that the input matrix
5501 `A` can itself be optionally transposed before the operation, which would cause either an assignment
5502 from `A` to `C` or an accumulation of `A` into `C`.

5503 Logically, this operation occurs in three steps:

5504 **Setup** The internal matrix and mask used in the computation are formed and their domains
5505 and dimensions are tested for compatibility.

5506 **Compute** The indicated computations are carried out.

5507 **Output** The result is written into the output matrix, possibly under control of a mask.

5508 Up to three matrix arguments are used in this `GrB_transpose` operation:

- 5509 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5510 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)
- 5511 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5512 The argument matrices and accumulation operator (if provided) are tested for domain compatibility
5513 as follows:

- 5514 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
5515 must be from one of the pre-defined types of Table 2.2.
- 5516 2. $\mathbf{D}(C)$ must be compatible with $\mathbf{D}(A)$ of the input matrix.
- 5517 3. If `accum` is not `GrB_NULL`, then $\mathbf{D}(C)$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
5518 of the accumulation operator and $\mathbf{D}(A)$ of the input matrix must be compatible with $\mathbf{D}_{in_2}(\text{accum})$
5519 of the accumulation operator.

5520 Two domains are compatible with each other if values from one domain can be cast to values in
 5521 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all
 5522 compatible with each other. A domain from a user-defined type is only compatible with itself. If
 5523 any compatibility rule above is violated, execution of `GrB_transpose` ends and the domain mismatch
 5524 error listed above is returned.

5525 From the argument matrices, the internal matrices and mask used in the computation are formed
 5526 (\leftarrow denotes copy):

- 5527 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 5528 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
 - 5529 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
 5530 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
 - 5531 (b) If `Mask \neq GrB_NULL`,
 - 5532 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
 5533 $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$,
 - 5534 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
 5535 $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$.
 - 5536 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 5537 3. Matrix $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB_INP0}].\mathbf{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.

5538 The internal matrices and masks are checked for dimension compatibility. The following conditions
 5539 must hold:

- 5540 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 5541 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 5542 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$.
- 5543 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$.

5544 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension
 5545 mismatch error listed above is returned.

5546 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5547 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5548 We are now ready to carry out the matrix transposition and any additional associated operations.
 5549 We describe this in terms of two intermediate matrices:

- 5550 • $\tilde{\mathbf{T}}$: The matrix holding the transpose of $\tilde{\mathbf{A}}$.
- 5551 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5552 The intermediate matrix

$$5553 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\tilde{\mathbf{A}}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{L}(\tilde{\mathbf{T}}) = \{(j, i, A_{ij}) \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$$

5554 is created.

5555 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 5556 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 5557 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5558 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5559 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
5560 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5561 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$
$$5562 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$
$$5563 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5564

5565

5566 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5567 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
5568 using what is called a *standard matrix mask and replace*. This is carried out under control of the
5569 mask which acts as a “write mask”.

- 5570 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
5571 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5572 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 5573 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
5574 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
5575 mask are unchanged:

$$5576 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5577 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content
5578 of matrix \mathbf{C} is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method
5579 exits with return value `GrB_SUCCESS` and the new content of matrix \mathbf{C} is as defined above but
5580 may not be fully computed. However, it can be used in the next GraphBLAS method call in a
5581 sequence.

5582 4.3.11 kronecker: Kronecker product of two matrices

5583 Computes the Kronecker product of two matrices. The result is a matrix.

5584 C Syntax

```
5585     GrB_Info GrB_kronecker(GrB_Matrix      C,  
5586                          const GrB_Matrix Mask,  
5587                          const GrB_BinaryOp accum,  
5588                          const GrB_Semiring op,  
5589                          const GrB_Matrix  A,  
5590                          const GrB_Matrix  B,  
5591                          const GrB_Descriptor desc);  
5592  
5593     GrB_Info GrB_kronecker(GrB_Matrix      C,  
5594                          const GrB_Matrix Mask,  
5595                          const GrB_BinaryOp accum,  
5596                          const GrB_Monoid  op,  
5597                          const GrB_Matrix  A,  
5598                          const GrB_Matrix  B,  
5599                          const GrB_Descriptor desc);  
5600  
5601     GrB_Info GrB_kronecker(GrB_Matrix      C,  
5602                          const GrB_Matrix Mask,  
5603                          const GrB_BinaryOp accum,  
5604                          const GrB_BinaryOp op,  
5605                          const GrB_Matrix  A,  
5606                          const GrB_Matrix  B,  
5607                          const GrB_Descriptor desc);
```

5608 Parameters

5609 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values
5610 that may be accumulated with the result of the Kronecker product. On output,
5611 the matrix holds the results of the operation.

5612 **Mask** (IN) An optional “write” mask that controls which results from this operation are
5613 stored into the output matrix C. The mask dimensions must match those of the
5614 matrix C. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain
5615 of the Mask matrix must be of type bool or any of the predefined “built-in” types
5616 in Table 2.2. If the default mask is desired (i.e., a mask that is all true with the
5617 dimensions of C), GrB_NULL should be specified.

5618 **accum** (IN) An optional binary operator used for accumulating entries into existing C
5619 entries. If assignment rather than accumulation is desired, GrB_NULL should be
5620 specified.

5621 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”
5622 operation. Depending on which type is passed, the following defines the binary
5623 operator, $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$, used:

5652 GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the
 5653 corresponding domains of the binary operator (`op`) or accumulation
 5654 operator, or the mask's domain is not compatible with `bool` (in the
 5655 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

5656 Description

5657 GrB_kronecker computes the Kronecker product $C = A \otimes B$ or, if an optional binary accumulation
 5658 operator (\odot) is provided, $C = C \odot (A \otimes B)$ (where matrices A and B can be optionally transposed).
 5659 The Kronecker product is defined as follows:

$$5660 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

5662 where $A : \mathbb{S}^{m_A \times n_A}$, $B : \mathbb{S}^{m_B \times n_B}$, and $C : \mathbb{S}^{m_A m_B \times n_A n_B}$. More explicitly, the elements of the
 5663 Kronecker product are defined as

$$5664 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

5665 where \otimes is the multiplicative operator specified by the `op` parameter.

5666 Logically, this operation occurs in three steps:

5667 **Setup** The internal matrices and mask used in the computation are formed and their domains
 5668 and dimensions are tested for compatibility.

5669 **Compute** The indicated computations are carried out.

5670 **Output** The result is written into the output matrix, possibly under control of a mask.

5671 Up to four argument matrices are used in the GrB_kronecker operation:

- 5672 1. $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5673 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)
- 5674 3. $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 5675 4. $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

5676 The argument matrices, the "product" operator (`op`), and the accumulation operator (if provided)
 5677 are tested for domain compatibility as follows:

- 5678 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then $\mathbf{D}(\text{Mask})$
 5679 must be from one of the pre-defined types of Table 2.2.

- 5680 2. $\mathbf{D}(\mathbf{A})$ must be compatible with $\mathbf{D}_{in_1}(\text{op})$.
- 5681 3. $\mathbf{D}(\mathbf{B})$ must be compatible with $\mathbf{D}_{in_2}(\text{op})$.
- 5682 4. $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{out}(\text{op})$.
- 5683 5. If `accum` is not `GrB_NULL`, then $\mathbf{D}(\mathbf{C})$ must be compatible with $\mathbf{D}_{in_1}(\text{accum})$ and $\mathbf{D}_{out}(\text{accum})$
- 5684 of the accumulation operator and $\mathbf{D}_{out}(\text{op})$ of `op` must be compatible with $\mathbf{D}_{in_2}(\text{accum})$ of
- 5685 the accumulation operator.

5686 Two domains are compatible with each other if values from one domain can be cast to values in

5687 the other domain as per the rules of the C language. In particular, domains from Table 2.2 are all

5688 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5689 any compatibility rule above is violated, execution of `GrB_kronecker` ends and the domain mismatch

5690 error listed above is returned.

5691 From the argument matrices, the internal matrices and mask used in the computation are formed

5692 (\leftarrow denotes copy):

- 5693 1. Matrix $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$.
- 5694 2. Two-dimensional mask, $\tilde{\mathbf{M}}$, is computed from argument `Mask` as follows:
- 5695 (a) If `Mask = GrB_NULL`, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
- 5696 $j < \mathbf{ncols}(\mathbf{C})\} \rangle$.
- 5697 (b) If `Mask \neq GrB_NULL`,
- 5698 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$
- 5699 $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$,
- 5700 ii. Otherwise, $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$
- 5701 $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$.
- 5702 (c) If `desc[GrB_MASK].GrB_COMP` is set, then $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$.
- 5703 3. Matrix $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB_INP0}].\text{GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$.
- 5704 4. Matrix $\tilde{\mathbf{B}} \leftarrow \text{desc}[\text{GrB_INP1}].\text{GrB_TRAN} ? \mathbf{B}^T : \mathbf{B}$.

5705 The internal matrices and masks are checked for dimension compatibility. The following conditions

5706 must hold:

- 5707 1. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$.
- 5708 2. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$.
- 5709 3. $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) \cdot \mathbf{nrows}(\tilde{\mathbf{B}})$.
- 5710 4. $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) \cdot \mathbf{ncols}(\tilde{\mathbf{B}})$.

5711 If any compatibility rule above is violated, execution of `GrB_kronecker` ends and the dimension
 5712 mismatch error listed above is returned.

5713 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with
 5714 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5715 We are now ready to carry out the Kronecker product and any additional associated operations.
 5716 We describe this in terms of two intermediate matrices:

- 5717 • $\tilde{\mathbf{T}}$: The matrix holding the Kronecker product of matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$.
- 5718 • $\tilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

5719 The intermediate matrix $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$
 5720 $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$ is created. The value of
 5721 each of its elements is computed by

$$5722 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

5723 where \otimes is the multiplicative operator specified by the `op` parameter.

5724 The intermediate matrix $\tilde{\mathbf{Z}}$ is created as follows, using what is called a *standard matrix accumulate*:

- 5725 • If `accum = GrB_NULL`, then $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$.
- 5726 • If `accum` is a binary operator, then $\tilde{\mathbf{Z}}$ is defined as

$$5727 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5728 The values of the elements of $\tilde{\mathbf{Z}}$ are computed based on the relationships between the sets of
 5729 indices in $\tilde{\mathbf{C}}$ and $\tilde{\mathbf{T}}$.

$$5730 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5731 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5732 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5733
 5734
 5735 where $\odot = \odot(\text{accum})$, and the difference operator refers to set difference.

5736 Finally, the set of output values that make up matrix $\tilde{\mathbf{Z}}$ are written into the final result matrix \mathbf{C} ,
 5737 using what is called a *standard matrix mask and replace*. This is carried out under control of the
 5738 mask which acts as a “write mask”.

- 5739 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in \mathbf{C} on input to this operation are
 5740 deleted and the content of the new output matrix, \mathbf{C} , is defined as,

$$5741 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5742 • If desc[GrB_OUTP].GrB_REPLACE is not set, the elements of $\tilde{\mathbf{Z}}$ indicated by the mask are
5743 copied into the result matrix, \mathbf{C} , and elements of \mathbf{C} that fall outside the set indicated by the
5744 mask are unchanged:

$$5745 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5746 In GrB_BLOCKING mode, the method exits with return value GrB_SUCCESS and the new content
5747 of matrix \mathbf{C} is as defined above and fully computed. In GrB_NONBLOCKING mode, the method
5748 exits with return value GrB_SUCCESS and the new content of matrix \mathbf{C} is as defined above but may
5749 not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.
5750 S

5751 4.4 Sequence Termination

5752 4.4.1 wait: Wait for pending operations to complete

5753 Waits for a collection of pending operations to complete. Two variants are supported, one that
5754 waits on all pending operations and one that waits on pending operations with a particular output
5755 object.

5756 4.4.1.1 wait: Waits until all pending operations complete variant

5757 When running in non-blocking mode, this function guarantees that all pending GraphBLAS opera-
5758 tions are fully executed. Note that this can be called in blocking mode without an error, but there
5759 should be no pending GraphBLAS operations to complete.

5760 C Syntax

```
5761         GrB_Info GrB_wait();
```

5762 Parameters

5763 Return values

5764 GrB_SUCCESS operation completed successfully.

5765 GrB_INDEX_OUT_OF_BOUNDS an index out-of-bounds execution error happened during comple-
5766 tion of pending operations.

5767 GrB_OUT_OF_MEMORY and out-of-memory execution error happened during completion of
5768 pending operations.

5769 GrB_PANIC unknown internal error.

5770 **Description**

5771 Upon successful return, all previously called GraphBLAS methods have fully completed their exe-
5772 cution, and any (transparent or opaque) data structures produced or manipulated by those methods
5773 can be safely touched. If an error occurred in any pending GraphBLAS operations, `GrB_error()` can
5774 be used to retrieve implementation defined error information about the problem encountered.

5775 **4.4.1.2 wait: Waits until pending operations on a specific object complete variant**

5776 When running in non-blocking mode, this function guarantees that all pending GraphBLAS oper-
5777 ations that have a specific GraphBLAS object as output are fully executed. Note that this can be
5778 called in blocking mode without an error, but there should be no pending GraphBLAS operations
5779 to complete.

5780 **C Syntax**

5781 `GrB_Info GrB_wait(GrB_Object *obj);`

5782 **Parameters**

5783 `obj` (IN) An existing GraphBLAS object. The object must have been created by an
5784 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS
5785 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,
5786 or type. On successful return of `GrB_wait`, all GraphBLAS operations that produce
5787 `obj` as output have fully completed.

5788 **Return values**

5789 `GrB_SUCCESS` operation completed successfully.

5790 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during comple-
5791 tion of pending operations.

5792 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion of
5793 pending operations.

5794 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`
5795 method.

5796 `GrB_PANIC` unknown internal error.

5797 **Description**

5798 Upon successful return, all previously called GraphBLAS methods that have `obj` as an `OUT` or
5799 `INOUT` parameter have fully completed their execution, and any (transparent or opaque) data

5800 structures produced or manipulated by those methods can be safely touched. If an error occurred
5801 in any of those GraphBLAS operations, `GrB_error()` can be used to retrieve implementation defined
5802 error information about the problem encountered.

5803 In non-blocking mode, a call to `GrB_wait(obj)` does not necessarily end the current GraphBLAS
5804 sequence. If there are other pending methods in the sequence, producing other objects, there is
5805 no guarantee that those methods have completed. Those methods can still produce errors and/or
5806 consume execution time.

5807 **4.4.2 error: Get an error message regarding internal errors**

```
5808     const char *GrB_error();
```

5809 **Parameters**

5810 **Return value**

- 5811 • A pointer to a null-terminated string (owned by the library).

5812 **Description**

5813 After a call to any GraphBLAS method, the program can retrieve additional error information
5814 (beyond the error code returned by the method) through a call to the function `GrB_error()`. The
5815 function returns a pointer to a null terminated string and the contents of that string are implemen-
5816 tation dependent. In particular, a null string (not a `NULL` pointer) is always a valid error string.
5817 The pointer is valid until the next call to any GraphBLAS method by the same thread. `GrB_error()`
5818 is a thread-safe function, in the sense that multiple threads can call it simultaneously and each will
5819 get its own error string back, referring to the last GraphBLAS method it called.

5820 **Chapter 5**

5821 **Nonpolymorphic Interface**

5822 Each polymorphic GraphBLAS method (those with multiple parameter signatures under the same
 5823 name) has a corresponding set of long-name forms that are specific to each parameter signature.
 5824 That is show in Tables 5.1 through 5.8.

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
<code>GrB_free(GrB_Type*)</code>	<code>GrB_Type_free(GrB_Type*)</code>
<code>GrB_free(GrB_UnaryOp*)</code>	<code>GrB_UnaryOp_free(GrB_UnaryOp*)</code>
<code>GrB_free(GrB_BinaryOp*)</code>	<code>GrB_BinaryOp_free(GrB_BinaryOp*)</code>
<code>GrB_free(GrB_Monoid*)</code>	<code>GrB_Monoid_free(GrB_Monoid*)</code>
<code>GrB_free(GrB_Semiring*)</code>	<code>GrB_Semiring_free(GrB_Semiring*)</code>
<code>GrB_free(GrB_Vector*)</code>	<code>GrB_Vector_free(GrB_Vector*)</code>
<code>GrB_free(GrB_Matrix*)</code>	<code>GrB_Matrix_free(GrB_Matrix*)</code>
<code>GrB_free(GrB_Descriptor*)</code>	<code>GrB_Descriptor_free(GrB_Descriptor*)</code>
<code>GrB_wait(GrB_Type*)</code>	<code>GrB_Type_wait(GrB_Type*)</code>
<code>GrB_wait(GrB_UnaryOp*)</code>	<code>GrB_UnaryOp_wait(GrB_UnaryOp*)</code>
<code>GrB_wait(GrB_BinaryOp*)</code>	<code>GrB_BinaryOp_wait(GrB_BinaryOp*)</code>
<code>GrB_wait(GrB_Monoid*)</code>	<code>GrB_Monoid_wait(GrB_Monoid*)</code>
<code>GrB_wait(GrB_Semiring*)</code>	<code>GrB_Semiring_wait(GrB_Semiring*)</code>
<code>GrB_wait(GrB_Vector*)</code>	<code>GrB_Vector_wait(GrB_Vector*)</code>
<code>GrB_wait(GrB_Matrix*)</code>	<code>GrB_Matrix_wait(GrB_Matrix*)</code>
<code>GrB_wait(GrB_Descriptor*)</code>	<code>GrB_Descriptor_wait(GrB_Descriptor*)</code>

Table 5.5: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
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,...)
GrB_extract(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_extract(GrB_Vector,...,GrB_Vector,...)
GrB_extract(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_extract(GrB_Matrix,...,GrB_Matrix,...)
GrB_extract(GrB_Vector,...,GrB_Matrix,...)	GrB_Col_extract(GrB_Vector,...,GrB_Matrix,...)
GrB_assign(GrB_Vector,...,GrB_Vector,...)	GrB_Vector_assign(GrB_Vector,...,GrB_Vector,...)
GrB_assign(GrB_Matrix,...,GrB_Matrix,...)	GrB_Matrix_assign(GrB_Matrix,...,GrB_Matrix,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)	GrB_Col_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...)
GrB_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)	GrB_Row_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)
GrB_assign(GrB_Vector,...,bool,...)	GrB_Vector_assign_BOOL(GrB_Vector,...,bool,...)
GrB_assign(GrB_Vector,...,int8_t,...)	GrB_Vector_assign_INT8(GrB_Vector,...,int8_t,...)
GrB_assign(GrB_Vector,...,uint8_t,...)	GrB_Vector_assign_UINT8(GrB_Vector,...,uint8_t,...)
GrB_assign(GrB_Vector,...,int16_t,...)	GrB_Vector_assign_INT16(GrB_Vector,...,int16_t,...)
GrB_assign(GrB_Vector,...,uint16_t,...)	GrB_Vector_assign_UINT16(GrB_Vector,...,uint16_t,...)
GrB_assign(GrB_Vector,...,int32_t,...)	GrB_Vector_assign_INT32(GrB_Vector,...,int32_t,...)
GrB_assign(GrB_Vector,...,uint32_t,...)	GrB_Vector_assign_UINT32(GrB_Vector,...,uint32_t,...)
GrB_assign(GrB_Vector,...,int64_t,...)	GrB_Vector_assign_INT64(GrB_Vector,...,int64_t,...)
GrB_assign(GrB_Vector,...,uint64_t,...)	GrB_Vector_assign_UINT64(GrB_Vector,...,uint64_t,...)
GrB_assign(GrB_Vector,...,float,...)	GrB_Vector_assign_FP32(GrB_Vector,...,float,...)
GrB_assign(GrB_Vector,...,double,...)	GrB_Vector_assign_FP64(GrB_Vector,...,double,...)
GrB_assign(GrB_Vector,...,other,...)	GrB_Vector_assign_UDT(GrB_Vector,...,const void*,...)
GrB_assign(GrB_Matrix,...,bool,...)	GrB_Matrix_assign_BOOL(GrB_Matrix,...,bool,...)
GrB_assign(GrB_Matrix,...,int8_t,...)	GrB_Matrix_assign_INT8(GrB_Matrix,...,int8_t,...)
GrB_assign(GrB_Matrix,...,uint8_t,...)	GrB_Matrix_assign_UINT8(GrB_Matrix,...,uint8_t,...)
GrB_assign(GrB_Matrix,...,int16_t,...)	GrB_Matrix_assign_INT16(GrB_Matrix,...,int16_t,...)
GrB_assign(GrB_Matrix,...,uint16_t,...)	GrB_Matrix_assign_UINT16(GrB_Matrix,...,uint16_t,...)
GrB_assign(GrB_Matrix,...,int32_t,...)	GrB_Matrix_assign_INT32(GrB_Matrix,...,int32_t,...)
GrB_assign(GrB_Matrix,...,uint32_t,...)	GrB_Matrix_assign_UINT32(GrB_Matrix,...,uint32_t,...)
GrB_assign(GrB_Matrix,...,int64_t,...)	GrB_Matrix_assign_INT64(GrB_Matrix,...,int64_t,...)
GrB_assign(GrB_Matrix,...,uint64_t,...)	GrB_Matrix_assign_UINT64(GrB_Matrix,...,uint64_t,...)
GrB_assign(GrB_Matrix,...,float,...)	GrB_Matrix_assign_FP32(GrB_Matrix,...,float,...)
GrB_assign(GrB_Matrix,...,double,...)	GrB_Matrix_assign_FP64(GrB_Matrix,...,double,...)
GrB_assign(GrB_Matrix,...,other,...)	GrB_Matrix_assign_UDT(GrB_Matrix,...,const void*,...)
GrB_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)	GrB_Vector_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)
GrB_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)	GrB_Matrix_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)

Table 5.6: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_BOOL(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_INT8(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_UINT8(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_INT16(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_UINT16(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_INT32(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_UINT32(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_INT64(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_UINT64(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_FP32(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_FP64(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,<i>other</i>,GrB_Vector,...)</code>	<code>GrB_Vector_apply_BinaryOp1st_UDT(GrB_Vector,...,GrB_BinaryOp,const void*,GrB_Vector,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_BOOL(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_INT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_UINT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_INT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_UINT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_INT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_UINT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_INT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_UINT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_FP32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_FP64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)</code>
<code>GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,<i>other</i>,...)</code>	<code>GrB_Vector_apply_BinaryOp2nd_UDT(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,const void*,...)</code>

Table 5.7: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_BOOL(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_INT8(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_UINT8(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_INT16(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_UINT16(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_INT32(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_UINT32(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_INT64(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_UINT64(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_FP32(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_FP64(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,<i>other</i>,GrB_Matrix,...)</code>	<code>GrB_Matrix_apply_BinaryOp1st_UDT(GrB_Matrix,...,GrB_BinaryOp,const void*,GrB_Matrix,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_BOOL(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_INT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_UINT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_INT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_UINT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_INT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_UINT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_INT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_UINT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_FP32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_FP64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)</code>
<code>GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,<i>other</i>,...)</code>	<code>GrB_Matrix_apply_BinaryOp2nd_UDT(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,const void*,...)</code>

Table 5.8: 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>
<code>GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)</code>	<code>GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)</code>
<code>GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)</code>	<code>GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)</code>
<code>GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)</code>	<code>GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)</code>

5825 Appendix A

5826 Revision History

5827 Changes in 1.3.0 (25 September 2019):

- 5828 • (Issue 50) Changed definition of completion and added `GrB_wait()` that takes an opaque
5829 GraphBLAS object as an argument.
- 5830 • (Issue 39) Added `GrB_kronecker` operation.
- 5831 • (Issue 40) Added variants of the `GrB_apply` operation that take a binary function and a scalar.
- 5832 • (Issue 59) Changed specification about how reductions to scalar (`GrB_reduce`) are to be per-
5833 formed (to minimize dependence on monoid identity).
- 5834 • (Issue 24) Added methods to resize matrices and vectors (`GrB_Matrix_resize` and `GrB_Vector_resize`).
- 5835 • (Issue 47) Added methods to remove single elements from matrices and vectors (`GrB_Matrix_removeElement`
5836 and `GrB_Vector_removeElement`).
- 5837 • (Issue 41) Added `GrB_STRUCTURE` descriptor flag for masks (consider only the structure of
5838 the mask and not the values).
- 5839 • (Issue 64) Deprecated `GrB_SCMP` in favor of new `GrB_COMP` for descriptor values.
- 5840 • (Issue 46) Added predefined descriptors covering all possible combinations of field, value pairs.
- 5841 • Added unary operators: absolute value (`GrB_ABS_I`) and bitwise complement of integers
5842 (`GrB_BNOT_I`).
- 5843 • (Issues 42,62) Added binary operators: Added boolean exclusive-nor (`GrB_LXNOR`) and bit-
5844 wise logical operators on integers (`GrB_BOR_I`, `GrB_BAND_I`, `GrB_BXOR_I`, `GrB_BXNOR_I`).
- 5845 • (Issue 11) Added a set of predefined monoids and semirings.
- 5846 • (Issue 57) Updated all examples in the appendix to take advantage of new capabilities and
5847 predefined objects.
- 5848 • (Issue 43) Added parent-BFS example.

- 5849 • (Issue 1) Fixed bug in the non-batch betweenness centrality algorithm in Appendix B.4 where
5850 source nodes were incorrectly assigned path counts.
- 5851 • (Issue 3) Added compile-time preprocessor defines and runtime method for querying the
5852 GraphBLAS API version being used.
- 5853 • (Issue 10) Clarified GrB_init() and GrB_finalize() errors.
- 5854 • (Issue 16) Clarified behavior of boolean and integer division.
- 5855 • (Issue 19) Clarified aliasing in user-defined operators.
- 5856 • (Issue 20) Clarified language about behavior of GrB_free() with predefined objects (implemen-
5857 tation defined)
- 5858 • (Issue 55) Clarified that multiplication does not have to distribute over addition in a Graph-
5859 BLAS semiring.
- 5860 • (Issue 45) Removed unnecessary language about annihilators.
- 5861 • (Issue 61) Removed unnecessary language about implied zeros.
- 5862 • (Issue 60) Added disclaimer against overspecification.
- 5863 • Fixed miscellaneous typographical errors (such as $\otimes \oplus$).

5864 Changes in 1.2.0:

- 5865 • Removed "provisional" clause.

5866 Changes in 1.1.0:

- 5867 • Removed unnecessary const from nindices, nrows, and ncols parameters of both extract and
5868 assign operations.
- 5869 • Signature of GrB_UnaryOp_new changed: order of input parameters changed.
- 5870 • Signature of GrB_BinaryOp_new changed: order of input parameters changed.
- 5871 • Signature of GrB_Monoid_new changed: removal of domain argument which is now inferred
5872 from the domains of the binary operator provided.
- 5873 • Signature of GrB_Vector_extractTuples and GrB_Matrix_extractTuples to add an in/out argu-
5874 ment, n, which indicates the size of the output arrays provided (in terms of number of ele-
5875 ments, not number of bytes). Added new execution error, GrB_INSUFFICIENT_SPACE which
5876 is returned when the capacities of the output arrays are insufficient to hold all of the tuples.
- 5877 • Changed GrB_Column_assign to GrB_Col_assign for consistency in non-polymorphic interface.
- 5878 • Added replace flag (z) notation to Table 4.1.

- 5879 • Updated the “Mathematical Description” of the assign operation in Table 4.1.
 - 5880 • Added triangle counting example.
 - 5881 • Added subsection headers for accumulate and mask/replace discussions in the Description
 - 5882 sections of GraphBLAS operations when the respective text was the “standard” text (i.e.,
 - 5883 identical in a majority of the operations).
 - 5884 • Fixed typographical errors.
- 5885 Changes in 1.0.2:
- 5886 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
 - 5887 matrices and avoid casting issues in certain implementations.
 - 5888 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
 - 5889 erased outside the assigned area.
 - 5890 • Changes non-polymorphic interface:
 - 5891 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.
 - 5892 – Renamed `GrB_Vector_reduce_BinaryOp` to `GrB_Matrix_reduce_BinaryOp`.
 - 5893 – Renamed `GrB_Vector_reduce_Monoid` to `GrB_Matrix_reduce_Monoid`.
 - 5894 • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
 - 5895 • Fixed numerous typographical errors.

5896 **Appendix B**

5897 **Examples**

B.1 Example: level 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     /*
24      * BFS traversal and label the vertices.
25      */
26     int32_t d = 0;                   //  $d =$  level in BFS traversal
27     bool succ = false;               //  $\text{succ} == \text{true}$  when some successor found
28     do {
29         ++d;                          // next level (start with 1)
30         GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL); //  $v[q] = d$ 
31         GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
32             q,A,GrB_DESC_RC);         //  $q[!v] = q \ || \ \&\& \ A$ ; finds all the
33                                     // unvisited successors from current  $q$ 
34         GrB_reduce(&succ,GrB_NULL,GrB_LOR_MONOID_BOOL,
35             q,GrB_NULL);              //  $\text{succ} = ||(q)$ 
36     } while (succ);                  // if there is no successor in  $q$ , we are done.
37
38     GrB_free(&q);                    //  $q$  vector no longer needed
39
40     return GrB_SUCCESS;
41 }

```

B.2 Example: level 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 /*
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  $v[i]$  does not have a stored element.
11 * Vector  $v$  should be uninitialized on input.
12 */
13 GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
14 {
15     GrB_Index n;
16     GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
17
18     GrB_Vector_new(v,GrB_INT32,n);    // Vector<int32_t>  $v(n) = 0$ 
19
20     GrB_Vector q;                    // vertices visited in each level
21     GrB_Vector_new(&q,GrB_BOOL,n);    // Vector<bool>  $q(n) = false$ 
22     GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = true$ , false everywhere else
23
24     /*
25      * BFS traversal and label the vertices.
26      */
27     int32_t level = 0;                // level = depth in BFS traversal
28     GrB_Index nvals;
29     do {
30         ++level;                      // next level (start with 1)
31         GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,
32                 GrB_SECOND_INT32,q,level,GrB_NULL); //  $v[q] = level$ 
33         GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
34                q,A,GrB_DESC_RC);      //  $q[!v] = q \ || \ \&\& \ A$ ; finds all the
35                                     // unvisited successors from current  $q$ 
36         GrB_Vector_nvals(&nvals, q);
37     } while (nvals);                  // if there is no successor in  $q$ , we are done.
38
39     GrB_free(&q);                      //  $q$  vector no longer needed
40
41     return GrB_SUCCESS;
42 }

```

B.3 Example: parent 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 binary  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS
9  * traversal of the graph and sets  $parents[i]$  to the index of vertex  $i$ 's parent.
10 * The parent of the root vertex,  $s$ , will be set to itself ( $parents[s] = s$ ). If
11 * vertex  $i$  is not reachable from  $s$ ,  $parents[i]$  will not contain a stored value.
12 */
13 GrB_Info BFS(GrB_Vector *parents, const GrB_Matrix A, GrB_Index s)
14 {
15     GrB_Index N;
16     GrB_Matrix_nrows(&N, A);           //  $N = \#$  vertices
17
18     // create index ramp for  $index\_of()$  functionality
19     GrB_Index *idx = (GrB_Index*)malloc(N*sizeof(GrB_Index));
20     for (GrB_Index i = 0; i < N; ++i) idx[i] = i;
21     GrB_Vector index_ramp;
22     GrB_Vector_new(&index_ramp, GrB_UINT64, N);
23     GrB_Vector_build_UINT64(index_ramp, idx, idx, N, GrB_PLUS_INT64);
24     free(idx);
25
26     GrB_Vector_new(parents, GrB_UINT64, N);
27     GrB_Vector_setElement(*parents, s, s);           //  $parents[s] = s$ 
28
29     GrB_Vector wavefront;
30     GrB_Vector_new(&wavefront, GrB_UINT64, N);
31     GrB_Vector_setElement(wavefront, 1UL, s);       //  $wavefront[s] = 1$ 
32
33     /*
34      * BFS traversal and label the vertices.
35      */
36     GrB_Index nvals;
37     GrB_Vector_nvals(&nvals, wavefront);
38
39     while (nvals > 0)
40     {
41         // convert all stored values in wavefront to their 0-based index
42         GrB_eWiseMult(wavefront, GrB_NULL, GrB_NULL, GrB_FIRST_UINT64,
43             index_ramp, wavefront, GrB_NULL);
44
45         // "FIRST" because left-multiplying wavefront rows. Masking out the parent
46         // list ensures wavefront values do not overwrite parents already stored.
47         GrB_vxm(wavefront, *parents, GrB_NULL, GrB_MIN_FIRST_SEMIRING_UINT64,
48             wavefront, A, GrB_DESC_RSC);
49
50         // Don't need to mask here since we did it in  $mxm$ . Merges new parents in
51         // current wavefront with existing parents:  $parents += wavefront$ 
52         GrB_apply(*parents, GrB_NULL, GrB_PLUS_UINT64,
53             GrB_IDENTITY_UINT64, wavefront, GrB_NULL);
54
55         GrB_Vector_nvals(&nvals, wavefront);
56     }
57
58     GrB_free(&wavefront);
59     GrB_free(&index_ramp);
60
61     return GrB_SUCCESS;
62 }
```


B.4 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_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
29             q, A, GrB_DESC_RC);      // get the first set of out neighbors
30
31     /*
32     * BFS phase
33     */
34     GrB_Index d = 0;                  // BFS level number
35     int32_t sum = 0;                  //  $\text{sum} == 0$  when BFS phase is complete
36
37     do {
38         GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\sigma[d, :] = q$ 
39         GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
40         GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
41                q, A, GrB_DESC_RC); //  $q = \#$  paths to nodes reachable
42                                     // from current level
43         GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
44         ++d;
45     } while (sum);
46
47     /*
48     * BC computation phase
49     *  $(t1, t2, t3, t4)$  are temporary vectors
50     */
51     GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
52     GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
53     GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
54     GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
55
56     for(int i=d-1; i>0; i--)
57     {
58         GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t1 = 1 + \delta$ 
59         GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_MONOID_FP32, t1, *delta, GrB_NULL);
60         GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t2 = \sigma[i, :]$ 
61         GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t2 = (1 + \delta) / \sigma[i, :]$ 
62         GrB_mxv(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32, // add contributions made by

```

```

63     A, t2, GrB_NULL);
64     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, GrB_DESC_T0); // t4 = sigma[i-1,:]
65     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, GrB_TIMES_FP32, t4, t3, GrB_NULL); // t4 = sigma[i-1,:]*t3
66     GrB_eWiseAdd(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, *delta, t4, GrB_NULL); // accumulate into delta
67 }
68
69 GrB_free(&sigma);
70 GrB_free(&q); GrB_free(&p);
71 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
72
73 return GrB_SUCCESS;
74 }

```

B.5 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 // index and value arrays needed to build numsp
12 GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
13 int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
14 for(int i=0; i<nsver; ++i) {
15     i_nsver[i] = i;
16     ones[i] = 1;
17 }
18
19 // numsp: structure holds the number of shortest paths for each node and starting vertex
20 // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
21 GrB_Matrix numsp;
22 GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
23 GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
24 free(i_nsver); free(ones);
25
26 // frontier: Holds the current frontier where values are path counts.
27 // Initialized to out vertices of each source node in s.
28 GrB_Matrix frontier;
29 GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
30 GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, GrB_DESC_RCT0);
31
32 // sigma: stores frontier information for each level of BFS phase. The memory
33 // for an entry in sigmas is only allocated within the do-while loop if needed.
34 // n is an upper bound on diameter.
35 GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n);
36
37 int32_t d = 0; // BFS level number
38 GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
39
40 // ----- The BFS phase (forward sweep) -----
41 do {
42     // sigmas[d](:,s) = dth level frontier from source vertex s
43     GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
44
45     GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
46              GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
47     GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL, GrB_PLUS_INT32
48                , numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
49     GrB_mxmx(frontier, numsp, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
50              A, frontier, GrB_DESC_RCT0); // f<!numsp> = A' +.* f (update frontier)
51     GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level
52     d++;
53 } while (nvals);
54
55 // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
56 GrB_Matrix nspinv;
57 GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
58 GrB_apply(nspinv, GrB_NULL, GrB_NULL,
59           GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
60
61 // bcu: BC updates for each vertex for each starting vertex in s
62 GrB_Matrix bcu;

```

```

63 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
64 GrB_assign(bcu, GrB_NULL, GrB_NULL,
65           1.0f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
66
67 GrB_Matrix w; // temporary workspace matrix
68 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
69
70 // ----- Tally phase (backward sweep) -----
71 for (int i=d-1; i>0; i--) {
72     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
73                 GrB_TIMES_FP32, bcu, nspinv, GrB_DESC_R); // w<sigmas[i]>=(1 ./ nsp).*bcu
74
75     // add contributions by successors and mask with that BFS level's frontier
76     GrB_mxm(w, sigmas[i-1], GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
77            A, w, GrB_DESC_R); // w<sigmas[i-1]> = (A +.* w)
78     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32, GrB_TIMES_FP32,
79                 w, numsp, GrB_NULL); // bcu += w .* numsp
80 }
81
82 // row reduce bcu and subtract "nsver" from every entry to account
83 // for 1 extra value per bcu row element.
84 GrB_reduce(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, bcu, GrB_NULL);
85 GrB_apply(*delta, GrB_NULL, GrB_NULL, GrB_MINUS_FP32, *delta, (float)nsver, GrB_NULL);
86
87 // Release resources
88 for (int i=0; i<d; i++) {
89     GrB_free(&(sigmas[i]));
90 }
91 free(sigmas);
92
93 GrB_free(&frontier); GrB_free(&numsp);
94 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
95
96 return GrB_SUCCESS;
97 }

```

B.6 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_UnaryOp set_random;
43     GrB_UnaryOp_new(&set_random, setRandom, GrB_FP32, GrB_UINT32);
44
45     // compute the degree of each vertex.
46     GrB_Vector degrees;
47     GrB_Vector_new(&degrees, GrB_FP64, n);
48     GrB_reduce(degrees, GrB_NULL, GrB_NULL, GrB_PLUS_FP64, A, GrB_NULL);
49
50     // Isolated vertices are not candidates: candidates[degrees != 0] = true
51     GrB_assign(candidates, degrees, GrB_NULL, true, GrB_ALL, n, GrB_NULL);
52
53     // add all singletons to iset: iset[degree == 0] = 1
54     GrB_assign(*iset, degrees, GrB_NULL, true, GrB_ALL, n, GrB_DESC_RC);
55
56     // Iterate while there are candidates to check.
57     GrB_Index nvals;
58     GrB_Vector_nvals(&nvals, candidates);
59     while (nvals > 0) {
60         // compute a random probability scaled by inverse of degree
61         GrB_apply(prob, candidates, GrB_NULL, set_random, degrees, GrB_DESC_R);
62     }

```

```

63 // compute the max probability of all neighbors
64 GrB_m xv(neighbor_max , candidates , GrB_NULL, GrB_MAX_SECOND.SEMIRING_FP32, A, prob , GrB_DESC_R);
65
66 // select vertex if its probability is larger than all its active neighbors ,
67 // and apply a "masked no-op" to remove stored falses
68 GrB_eWiseAdd( new_members , GrB_NULL, GrB_NULL, GrB_GT_FP64, prob , neighbor_max , GrB_NULL);
69 GrB_apply( new_members , new_members , GrB_NULL, GrB_IDENTITY_BOOL, new_members , GrB_DESC_R);
70
71 // add new members to independent set .
72 GrB_eWiseAdd( *iset , GrB_NULL, GrB_NULL, GrB_LOR, *iset , new_members , GrB_NULL);
73
74 // remove new members from set of candidates  $c = c \& \text{!new}$ 
75 GrB_eWiseMult( candidates , new_members , GrB_NULL,
76               GrB_LAND, candidates , candidates , GrB_DESC_RC);
77
78 GrB_Vector_nvals(&nvals , candidates);
79 if (nvals == 0) { break; } // early exit condition
80
81 // Neighbors of new members can also be removed from candidates
82 GrB_m xv( new_neighbors , candidates , GrB_NULL, GrB_LOR_LAND.SEMIRING_BOOL,
83         A, new_members , GrB_NULL);
84 GrB_eWiseMult( candidates , new_neighbors , GrB_NULL, GrB_LAND,
85               candidates , candidates , GrB_DESC_RC);
86
87 GrB_Vector_nvals(&nvals , candidates);
88 }
89
90 GrB_free(&neighbor_max); // free all objects "new'ed"
91 GrB_free(&new_members);
92 GrB_free(&new_neighbors);
93 GrB_free(&prob);
94 GrB_free(&candidates);
95 GrB_free(&set_random);
96 GrB_free(&degrees);
97
98 return GrB_SUCCESS;
99 }

```

B.7 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_mxm(C, L, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_UINT64, L, L, GrB_DESC_T1); // C<L> = L +.* L'
20
21     uint64_t count;
22     GrB_reduce(&count, GrB_NULL, GrB_PLUS_MONOID_UINT64, C, GrB_NULL); // 1-norm of C
23
24     GrB_free(&C);                               // C matrix no longer needed
25
26     return count;
27 }
```