The GraphBLAS C API Specification:
GrB_mxm() Review draft, Version 0.9.1

The GraphBLAS Signatures Subgroup
The GraphBLAS Forum

Generated on 2017/01/13 at 12:49:28 EDT
# Contents

## 1 Introduction

1. **2 Basic Concepts**
   - 2.1 Glossary
   - 2.2 Algebraic and Arithmetic foundations
   - 2.3 Domains
   - 2.4 Operators and Associated Functions
   - 2.5 Indices, Index Arrays and Scalar Arrays
   - 2.6 Execution Model
     - 2.6.1 Execution modes
     - 2.6.2 Thread safety
   - 2.7 Error Model

## 3 Objects

3.1 Operators
3.2 Monoids
3.3 Semirings
3.4 Vectors
3.5 Matrices
3.6 Masks
3.7 Descriptors

## 4 Methods
4.1 Context Methods ................................. 25
  4.1.1 init: Initialize a GraphBLAS context .................. 25
  4.1.2 finalize: Finalize a GraphBLAS context ............. 26

4.2 Object Methods ................................. 26
  4.2.1 Algebra Methods ............................. 26
  4.2.2 Vector Methods ............................. 31
  4.2.3 Matrix Methods ............................. 33
  4.2.4 Descriptor Methods ......................... 37
  4.2.5 free: Destroy object ....................... 38

4.3 GraphBLAS Operations ......................... 39
  4.3.1 buildMatrix: Store elements from tuples into a matrix .... 41
  4.3.2 buildVector: Store elements from tuples into a vector .... 42
  4.3.3 extractTuples: Extract tuples from a matrix .......... 44
  4.3.4 extractTuples: Extract tuples from a vector .......... 45
  4.3.5 mxm: Matrix-matrix multiply ................ 47

4.4 Sequence Termination ......................... 51
  4.4.1 wait: Waits until pending operations complete .... 51
Acknowledgments

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

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

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

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

- the Department of Energy Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231
- Intel Corporation
- Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University. [DM-0003727]
- International Business Machines Corporation
- Department of Defense under contract No. W911QX-12-C-0059, L-3 Data Tactics subcontract SCT-14-004 with University of California, Davis
This document is a preliminary draft of the GraphBLAS C standard. It includes text from the specification up to and including the matrix multiply operation \texttt{GrB.mxm()}. We plan to follow the template established by \texttt{GrB.mxm()} as we document the other GraphBLAS operations. Timely feedback on \texttt{GrB.mxm()} is therefore critical; hence why we’ve released this preliminary version of the document.
Chapter 1

Introduction

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

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

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

The remainder of this document is organized as follows:

- Chapter 2: Basic Concepts
- Chapter 3: Objects
- Chapter 4: Methods
Chapter 2

Basic Concepts

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

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

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

2.1 Glossary

- **application**: A program that calls methods from the GraphBLAS C API to solve a problem.
- **context**: The GraphBLAS C API was designed with the expectation that implementations of the API might utilize sophisticated execution strategies to optimize performance. We anticipate that an application may need to apply constraints on the behavior of implementations of the GraphBLAS C API and that these constraints may vary from one application to another. To the application, it appears as if the application interacts with a distinct instance of the GraphBLAS C API implementation. The state of this instance is called the GraphBLAS context. The context is set with `GrB_init()`. Currently, the only supported constraints on a context pertain to the mode of program execution.
- **domain**: The set of valid values for the elements of a GraphBLAS object.
- **function**: The term *function* as used in the GraphBLAS specification refers to a named group of statements in the C programming language. Methods, operators, and user-defined functions are typically implemented as C functions and when referring to the code programmers write as opposed to their role as an element of the GraphBLAS, they may be referred to as such.

- **GraphBLAS operators**: Binary or unary operators that act on elements of GraphBLAS objects. *GraphBLAS operators* are used to express algebraic structures used in the GraphBLAS such as monoids and semirings. *GraphBLAS operators* come in two types: (1) predefined operators found in Table 2.2 and (2) user-defined operators using `GrB_UnaryOp_new()` or `GrB_BinaryOp_new()` (see Section 4.2.1).

- **GraphBLAS object**: A instance of a data structure defined by the GraphBLAS C API that is opaque and manipulated only through the API. The fact the object is based on an opaque datatype gives an implementation of the GraphBLAS C API flexibility to optimize the structure for a particular platform.

- **GraphBLAS operation**: A mathematical operation defined in the GraphBLAS mathematical specification. These operations (not to be confused with *operators*) typically act on matrices and vectors with elements defined in terms of an algebraic semiring.

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

- **method**: A C function defined in the GraphBLAS C API that manipulates an opaque GraphBLAS object.

- **program order**: The text of an application program defines an order of the GraphBLAS methods. This order is called the *program order*.

- **sequence**: A series of GraphBLAS method calls in program order that: (1) begins with the first GraphBLAS method that creates or modifies a GraphBLAS object; and (2) terminates with a GraphBLAS method that explicitly waits for the sequence of methods to complete or any method that reads values from a GraphBLAS object into a non-opaque object.

- **mode**: A GraphBLAS sequence executes in one of two modes. In *blocking mode*, GraphBLAS methods return after the computations complete and any output objects have been updated. In nonblocking mode, a method returns once the arguments are tested as consistent with the method but potentially before any computation has proceeded.

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

- **non-opaque**: Any datatype that exposes its internal structure. This contrasts with an *opaque* datatype which hides its internals structure and can only be manipulated through an API.

- **semiring**: An algebraic structure consisting of a set of allowed values (the *domain*), two commutative binary operators called *addition* and *multiplication* over which the distributive

---

1We expect implementations of the GraphBLAS C API to use floating point arithmetic such as that defined by the IEEE-754 standard. Floating point arithmetic is not strictly associative.
property holds, identities over addition (0) and multiplication (1), and finally the additive identity is an annihilator over multiplication.

- **structural zero**: Also known as an implied zero, a structural zero is any element which has a valid index (or indices) in a GraphBLAS vector or matrix but is not explicitly identified in the list of elements of that vector or matrix. From a mathematical perspective, a structural zero is treated as having the value of the zero element of the relevant monoid or semiring.

- **structural complement**: The structural complement of a GraphBLAS vector or matrix of any domain is another vector or matrix, of domain bool, in which the explicitly identified elements (with a value of true) are the structural zeroes of the original vector or matrix (which have an implied value of false).

- **thread safe**: A routine is said to be thread safe if it performs its intended function even when executed concurrently (by more than one thread).

### 2.2 Algebraic and Arithmetic foundations

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

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

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

The mathematical formalism for graph operations in the language of linear algebra assumes that we can operate in the field of real numbers. However, the GraphBLAS C binding is designed for implementation on computers which by necessity have a finite number of bits to represent numbers. Therefore, we require a conforming implementation to use floating-point numbers such as those defined by the IEEE-754 standard (both single- and double-precision) wherever real numbers need to be represented. The practical implications of these finite precision numbers is that the result of a sequence of computations may vary from one execution to the next as the way operations are
Table 2.1: Predefined \texttt{GrB.Type} values, the corresponding C type (for scalar parameters, and domains for GraphBLAS.

<table>
<thead>
<tr>
<th>\texttt{GrB.Type} values</th>
<th>C type</th>
<th>domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{GrB_BOOL}</td>
<td>bool</td>
<td>{false, true}</td>
</tr>
<tr>
<td>\texttt{GrB_INT8}</td>
<td>int8_t</td>
<td>$\mathbb{Z} \cap [-2^7, 2^7)$</td>
</tr>
<tr>
<td>\texttt{GrB_UINT8}</td>
<td>uint8_t</td>
<td>$\mathbb{Z} \cap [0, 2^8)$</td>
</tr>
<tr>
<td>\texttt{GrB_INT16}</td>
<td>int16_t</td>
<td>$\mathbb{Z} \cap [-2^{15}, 2^{15})$</td>
</tr>
<tr>
<td>\texttt{GrB_UINT16}</td>
<td>uint16_t</td>
<td>$\mathbb{Z} \cap [0, 2^{16})$</td>
</tr>
<tr>
<td>\texttt{GrB_INT32}</td>
<td>int32_t</td>
<td>$\mathbb{Z} \cap [-2^{31}, 2^{31})$</td>
</tr>
<tr>
<td>\texttt{GrB_UINT32}</td>
<td>uint32_t</td>
<td>$\mathbb{Z} \cap [0, 2^{32})$</td>
</tr>
<tr>
<td>\texttt{GrB_INT64}</td>
<td>int64_t</td>
<td>$\mathbb{Z} \cap [-2^{63}, 2^{63})$</td>
</tr>
<tr>
<td>\texttt{GrB_UINT64}</td>
<td>uint64_t</td>
<td>$\mathbb{Z} \cap [0, 2^{64})$</td>
</tr>
<tr>
<td>\texttt{GrB_FLOAT}</td>
<td>float</td>
<td>IEEE 754 binary32</td>
</tr>
<tr>
<td>\texttt{GrB.DOUBLE}</td>
<td>double</td>
<td>IEEE 754 binary64</td>
</tr>
</tbody>
</table>

associated change. While techniques are known to reduce these effects, we do not require or even expect an implementation to use them as they may add considerable overhead. The fact is that in most cases, these roundoff errors are not significant and when they are significant, the problem itself is ill-conditioned and needs to be reformulated.

2.3 Domains

GraphBLAS defines two kinds of collections: matrices and vectors. For any given collection, the elements of the collection belong to a \textit{domain}, which is the set of valid values for the elements. In GraphBLAS, domains correspond to the valid values for types from the host language (in our case, the C programming language). For any variable or object $V$ in GraphBLAS we denote as $D(V)$ the domain of $V$; that is, the set of possible values that elements of $V$ can take. The predefined types and corresponding domains used in the GraphBLAS are shown in Table 2.1. The Boolean type is defined in \texttt{stdbool.h}, the integral types are defined in \texttt{stdint.h}, and the floating-point types are native to the language and in most cases defined by the IEEE-754 standard.

2.4 Operators and Associated Functions

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

Similar to GraphBLAS types having predefined types and user-defined types, GraphBLAS operators
come in two types as well: (1) predefined operators found in Table 2.2 and (2) user-defined operators using GrB_UnaryOp_new() or GrB_BinaryOp_new() (see Section 4.2.1).

2.5 Indices, Index Arrays and Scalar Arrays

In order to interface with third-party software packages, operations such as buildMatrix (§ 4.3.1) and extractTuples (§ 4.3.4) must specify how the data should be laid out in non-opaque data structures. To this end we define, explicitly 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:

```c
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.

2.6 Execution Model

A program using the GraphBLAS C API constructs GraphBLAS objects, manipulates them to implement a graph algorithm, and then extracts values from the GraphBLAS objects as the result of the algorithm. Functions defined within the GraphBLAS C API that manipulate GraphBLAS objects are called methods.

Graph algorithms are expressed as an ordered collection of GraphBLAS method calls defined by the order they are encountered in a program. This is called the Program Order. Each method in the collection uniquely and unambiguously defines the output GraphBLAS objects based on the GraphBLAS operation and the input GraphBLAS objects.

We define a sequence of GraphBLAS method calls, or when the meaning is clear a sequence, as a well defined ordered collection of GraphBLAS method calls. The initiation of a sequence is the first method call that modifies a GraphBLAS object, The end of the sequence is either (1) the first GraphBLAS method that reads values from a GraphBLAS object into a non-opaque data structure or (2) a GraphBLAS wait method. We collectively refer to these methods as terminating methods.

The set of operations between the initiation of a sequence and its termination mathematically define the result of that sequence.

2.6.1 Execution modes

A program using the GraphBLAS C API defines a collection of sequences. The execution model implied by these sequences depends on the mode of the GraphBLAS program. There are two
Table 2.2: Predefined unary and binary operators for GraphBLAS in C.

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

<table>
<thead>
<tr>
<th>Suffix</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>bool</td>
</tr>
<tr>
<td>I8</td>
<td>int8_t</td>
</tr>
<tr>
<td>U8</td>
<td>uint8_t</td>
</tr>
<tr>
<td>I16</td>
<td>int16_t</td>
</tr>
<tr>
<td>U16</td>
<td>uint16_t</td>
</tr>
<tr>
<td>I32</td>
<td>int32_t</td>
</tr>
<tr>
<td>U32</td>
<td>uint32_t</td>
</tr>
<tr>
<td>I64</td>
<td>int64_t</td>
</tr>
<tr>
<td>U64</td>
<td>uint64_t</td>
</tr>
<tr>
<td>F32</td>
<td>float</td>
</tr>
<tr>
<td>F64</td>
<td>double</td>
</tr>
</tbody>
</table>

(b) Predefined Operators.

<table>
<thead>
<tr>
<th>Operator type</th>
<th>GraphBLAS identifier</th>
<th>Domains</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_UnaryOp</td>
<td>GrB_LNOT</td>
<td>bool → bool</td>
<td>logical inverse</td>
</tr>
<tr>
<td>GrB_UnaryOp</td>
<td>GrB_SCMP</td>
<td>bool → bool</td>
<td>structural complement</td>
</tr>
<tr>
<td>GrB_UnaryOp</td>
<td>GrB_IDENTITY_T</td>
<td>$T \rightarrow T$</td>
<td>identity</td>
</tr>
<tr>
<td>GrB_UnaryOp</td>
<td>GrB_NEG_T</td>
<td>$T \rightarrow T$</td>
<td>arithmetic negation</td>
</tr>
<tr>
<td>GrB_UnaryOp</td>
<td>GrB_INV_T</td>
<td>$T \rightarrow T$</td>
<td>arithmetic inverse</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_LAND</td>
<td>bool × bool → bool</td>
<td>logical AND</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_LOR</td>
<td>bool × bool → bool</td>
<td>logical OR</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_LXOR</td>
<td>bool × bool → bool</td>
<td>logical XOR</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_FIRST_T</td>
<td>$T \times T \rightarrow T$</td>
<td>first argument</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_SECOND_T</td>
<td>$T \times T \rightarrow T$</td>
<td>second argument</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_MIN_T</td>
<td>$T \times T \rightarrow T$</td>
<td>minimum</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_MAX_T</td>
<td>$T \times T \rightarrow T$</td>
<td>maximum</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_PLUS_T</td>
<td>$T \times T \rightarrow T$</td>
<td>addition</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_MINUS_T</td>
<td>$T \times T \rightarrow T$</td>
<td>subtraction</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_TIMES_T</td>
<td>$T \times T \rightarrow T$</td>
<td>multiplication</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_DIV_T</td>
<td>$T \times T \rightarrow T$</td>
<td>division</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_EQ_T</td>
<td>$T \times T \rightarrow bool$</td>
<td>equal</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_NE_T</td>
<td>$T \times T \rightarrow bool$</td>
<td>not equal</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_GT_T</td>
<td>$T \times T \rightarrow bool$</td>
<td>greater than</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_LT_T</td>
<td>$T \times T \rightarrow bool$</td>
<td>less than</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_GE_T</td>
<td>$T \times T \rightarrow bool$</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>GrB_LE_T</td>
<td>$T \times T \rightarrow bool$</td>
<td>less than or equal</td>
</tr>
</tbody>
</table>
modes: blocking and nonblocking.

- **blocking**: In blocking mode, each method in a sequence completes the GraphBLAS operation defined by the method before proceeding to the next statement in program order. Output GraphBLAS objects defined by a method are stored in memory and are available to other C functions after each method returns.

- **nonblocking**: In nonblocking mode, each method may return once the input arguments have been inspected and verified to define a well-formed GraphBLAS operation. The GraphBLAS operation and the state of any GraphBLAS objects are undefined when a method returns until the terminating method in the sequence returns.

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

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

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

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

The mode is defined in the GraphBLAS C API when the context of the library invocation is defined. This occurs once before any GraphBLAS methods are called with a call to the `GrB_init()` function. After all GraphBLAS methods are complete, the context is terminated with a call to `GrB_finalize()`.

In the current version of the GraphBLAS C API, the context can only be set once in the execution of a program; i.e., after `GrB_finalize()` is called a following call to `GrB_init()` is not allowed.

### 2.6.2 Thread safety

Implementations of the GraphBLAS C API are required to be thread safe. Different threads may create GraphBLAS sequences that do not conflict and expect the results to be the same (within floating point roundoff errors) regardless of whether the sequences execute serially or concurrently.
Sequences that do not conflict are free of data races. A data race occurs when: (1) two or more threads access shared objects, (2) those access operations include at least one modify operation, and (3) those operations are not ordered through synchronization operations. The GraphBLAS C API does not provide synchronization operations to define ordered accesses to GraphBLAS objects. Hence the only way to assure that two sequences running concurrently on different threads do not conflict is if neither sequence writes to an object that the other sequence either reads or writes.

2.7 Error Model

All GraphBLAS methods return a value of type `GrB_info` to provide information available to the system at the time the method returns. In blocking mode, that information pertains to the full computation and the return values defined for each method in the specification provide information concerning the condition of the computation.

In nonblocking mode, the information pertains to a consistency check of the arguments to the method. Any method that terminates a sequence must return information about the status of that sequence of method calls. A return value of `GrB_SUCCESS` indicates that the method returned correctly and that the sequence produced the result defined by the sequence of GraphBLAS operations. Other return values from the method indicate that an error was found during execution of the sequence. When possible, that return value will provide information concerning the cause of the error. Additional information is returned in the null terminated character string, `err` which is always the last argument to any method that may terminate a sequence.

The defined error values are shown in Table 2.3. The errors fall into two groups: API errors (Table 2.3(a)) and execution errors (Table 2.3(b)). An API error means a GraphBLAS method was called with parameters that violate the rules for that method. API errors are deterministic and consistent across platforms and implementations. Execution errors indicate that something went wrong during the execution of a legal GraphBLAS method invocation. Their occurrence may depend on specifics of the executing environment. This does not mean that environment errors are the fault of the GraphBLAS implementation. For example, a memory leak is a program error but it may manifest itself in different points of program execution (or not at all) depending on the platform, problem size, or what else is running at that time.

If a GraphBLAS method returns with an API error, it is guaranteed that none of the method arguments (or any other program data) have been modified. If a GraphBLAS method returns with a `GrB_OUTOFMEM` error, it is guaranteed that no argument used as input-only has been modified. Output arguments may be left in an illegal state. Finally, if a GraphBLAS method returns with a `GrB_PANIC`, no guarantees can be made about the state of any program data.
Table 2.3: Error values returned by GraphBLAS methods.

(a) API errors

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_DOMAIN_MISMATCH</td>
<td></td>
</tr>
<tr>
<td>GrB_DIMENSION_MISMATCH</td>
<td></td>
</tr>
</tbody>
</table>

(b) Execution errors

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_OUTOFMEM</td>
<td>Not enough memory for operations</td>
</tr>
<tr>
<td>GrB_PANIC</td>
<td>Unknown internal error</td>
</tr>
</tbody>
</table>
Chapter 3

Objects

The following algebraic objects (operators, monoids, and semirings) are presented in increasing
generality. The “algebra generality rule” of GraphBLAS states that a more general object can
always be passed to any method which requires a less general object. The restriction rules are
explained in the respective sections of those objects.

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 do modify how algebraic objects operate on collections. More concretely,
descriptors can be used (among other things) to perform multiplication with transpose of matrix
without the user having to manually transpose the collection. A complete list of what descriptors
are capable of can be found in the section.

3.1 Operators

A GraphBLAS binary operators \( F_b = (D_1, D_2, D_3, \circ) \) is defined by three domains, \( D_1, D_2, D_3 \),
and an operation \( \circ: D_1 \times D_2 \rightarrow D_3 \). For a given GraphBLAS operators \( F_b = (D_1, D_2, D_3, \circ) \) we
define \( D_1(F_b) = D_1 \), \( D_2(F_b) = D_2 \), \( D_3(F_b) = D_3 \), and \( \bigcirc(F_b) = \circ \). Note that \( \circ \) could be used in
place of either \( \oplus \) or \( \odot \).

A GraphBLAS unary operators \( F_u = (D_1, D_2, f) \) is defined by two domains, \( D_1, D_2 \), and an
operation \( f: D_1 \rightarrow D_2 \). For a given GraphBLAS operators \( F_u = (D_1, D_2, f) \) we define \( D_1(F_u) = \)
\( D_1 \), \( D_2(F_u) = D_2 \), and \( f(F) = f \).

3.2 Monoids

A GraphBLAS generalized monoid (or monoid for short) \( M = (D_1, \circ, 0) \) is defined by a single
domain \( D_1 \), an associative operation \( \circ: D_1 \times D_1 \rightarrow D_1 \), and an identity element \( 0 \in D_1 \). For a

\(^1\)It is expected that implementations will utilize IEEE-754 floating point arithmetic which is not strictly associative.
Table 3.1: Properties and recipes for building GraphBLAS algebraic objects: Unary Operator, Binary Operator, Monoid and Semiring (composed of operations Add and Times).

Note 1: Output domain of Semiring Times must be same as domain of Semiring Add. This ensures 3 domains in total for Semiring rather than 4.

(a) Properties of algebraic objects.

<table>
<thead>
<tr>
<th>Object</th>
<th>Must Be Associative</th>
<th>Identity Must Exist</th>
<th>Number of Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary Operator</td>
<td>no</td>
<td>no</td>
<td>2</td>
</tr>
<tr>
<td>Binary Operator</td>
<td>no</td>
<td>no</td>
<td>3</td>
</tr>
<tr>
<td>Monoid</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Semiring Add</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Semiring Times</td>
<td>no</td>
<td>no</td>
<td>3 (Note 1)</td>
</tr>
</tbody>
</table>

(b) Recipes for algebraic objects.

<table>
<thead>
<tr>
<th>Object</th>
<th>Recipe</th>
<th>Number of Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary Operator</td>
<td>Function Pointer</td>
<td>2</td>
</tr>
<tr>
<td>Binary Operator</td>
<td>Function Pointer</td>
<td>3</td>
</tr>
<tr>
<td>Monoid</td>
<td>Associative Binary Operator with Identity</td>
<td>1</td>
</tr>
<tr>
<td>Semiring</td>
<td>Associative Binary Operator with Identity + Binary Operator</td>
<td>3</td>
</tr>
</tbody>
</table>
Table 3.2: Proposed operator input for relevant GraphBLAS operations. The semiring add and times are shown if applicable.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxm, mxv, vxm</td>
<td>Semiring</td>
</tr>
<tr>
<td>eWiseAdd</td>
<td>Binary Operator</td>
</tr>
<tr>
<td></td>
<td>Monoid</td>
</tr>
<tr>
<td></td>
<td>Semiring</td>
</tr>
<tr>
<td>eWiseMult</td>
<td>Binary Operator</td>
</tr>
<tr>
<td></td>
<td>Monoid</td>
</tr>
<tr>
<td></td>
<td>Semiring</td>
</tr>
<tr>
<td>reduce (to vector)</td>
<td>Binary Operator</td>
</tr>
<tr>
<td></td>
<td>Monoid</td>
</tr>
<tr>
<td>reduce (to scalar)</td>
<td>Monoid</td>
</tr>
<tr>
<td>apply</td>
<td>Unary Operator</td>
</tr>
<tr>
<td>buildMatrix (dups)</td>
<td>Binary Operator</td>
</tr>
<tr>
<td></td>
<td>Monoid</td>
</tr>
<tr>
<td>accum param, any op</td>
<td>Binary Operator</td>
</tr>
<tr>
<td></td>
<td>Monoid</td>
</tr>
</tbody>
</table>

354 given GraphBLAS monoid $M = \langle D_1, \odot, 0 \rangle$ we define $D_1(M) = D_1$, $\odot(M) = \odot$ and $0(M) = 0$. A GraphBLAS monoid is equivalent to the conventional monoid algebraic structure.

355 Let $F = \langle D_1, D_1, D_1, \odot \rangle$ be a GraphBLAS binary operator with element $0 \in D_1$. Then $M = \langle F, 0 \rangle = \langle D_1, \odot, 0 \rangle$ is a GraphBLAS monoid.

3.3 Semirings

358 A GraphBLAS semiring (or semiring for short) $S = \langle D_1, D_2, D_3, \oplus, \otimes, 0 \rangle$ is defined by three domains $D_1$, $D_2$ and $D_3$, an associative\(2\) additive operation $\oplus : D_1 \times D_2 \rightarrow D_3$, a multiplicative operation $\otimes : D_1 \times D_2 \rightarrow D_3$, and an element $0 \in D_3$. For a given GraphBLAS semiring $S = \langle D_1, D_2, D_3, \oplus, \otimes, 0 \rangle$ we define $D_1(S) = D_1$, $D_2(S) = D_2$, $D_3(S) = D_3$, $\oplus(S) = \oplus$, $\otimes(S) = \otimes$, and $0(S) = 0$.

359 Let $F = \langle D_1, D_2, D_3, \otimes \rangle$ be a operator and let $A = \langle D_3, \oplus, 0 \rangle$ be a monoid, then $S = \langle A, F \rangle = \langle D_1, D_2, D_3, \oplus, \otimes, 0 \rangle$ is a semiring.

360 Note: There must be one GraphBLAS monoid in every semiring which serves as the semiring’s additive operator and specifies the same domain for its inputs and output parameters.

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

\(2\)It is expected that implementations will utilize IEEE-754 floating point arithmetic which is not strictly associative.
3.4 Vectors

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

3.5 Matrices

A matrix \( \mathbf{A} = (D, M, N, \{(i, j, A_{ij})\}) \) is defined by a domain \( D \), its number of rows \( M > 0 \), its number of columns \( N > 0 \) and a set of tuples \( (i, j, A_{ij}) \) where \( 0 \leq i < M \), \( 0 \leq j < N \), and \( A_{ij} \in D \). A particular pair of values \( i, j \) can only appear at most once in \( \mathbf{A} \). We define \( \text{ncols}(\mathbf{A}) = N \), \( \text{nrows}(\mathbf{A}) = M \) and \( \text{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \). The set \( \text{L}(\mathbf{A}) \) is called the content of matrix \( \mathbf{A} \). We also define the sets \( \text{indrow}(\mathbf{A}) = \{i : \exists (i, j, A_{ij}) \in \mathbf{A}\} \) and \( \text{indcol}(\mathbf{A}) = \{j : \exists (i, j, A_{ij}) \in \mathbf{A}\} \). (These are the sets of nonempty rows and columns of \( \mathbf{A} \), respectively.) The structure of matrix \( \mathbf{A} \) is the set \( \text{ind}(\mathbf{A}) = \{(i, j) : (i, j, A_{ij}) \in \text{L}(\mathbf{A})\} \), and \( \text{D}(\mathbf{A}) = D \). For a matrix \( \mathbf{A} \), \( \mathbf{A}(i, j) \) is a reference to...
If \( A \) is a matrix and \( 0 \leq j < N \), then \( A(:,j) = \{ (i, A_{ij}) : (i, j, A) \in L(A) \} \) is a vector called the \( j \)-th column of \( A \). Correspondingly, if \( A \) is a matrix and \( 0 \leq i < M \), then \( A(i,:) = \{ (j, A_{ij}) : (i, j, A) \in L(A) \} \) is a vector called the \( i \)-th row of \( A \).

Given a matrix \( A = \{ (i, j, A) \} \), its transpose is another matrix \( A^T = \{ (j, i, A) : (i, j, A) \in L(A) \} \).

### 3.6 Masks

A mask can be either a one- or a two-dimensional construct. One- and two-dimensional masks, described more formally below, are similar to vectors and matrices, respectively, except that they have structure (indices) but no values. Masks are used to perform fine-grain control and optimization of GraphBLAS operations.

A one-dimensional mask \( m = \{ i \} \) is defined by its number of elements \( N > 0 \) and a set \( L(m) \) of indices \( \{ i \} \) where \( 0 \leq i < N \). A particular value of \( i \) can only appear at most once in \( m \). We define \( n(m) = N \). We also define the set \( \text{ind}(m) = \{ i : i \in L(m) \} \).

A two-dimensional mask \( M = \{ (i, j) \} \), is defined by its number of rows \( M > 0 \), its number of columns \( N > 0 \) and a set \( L(M) \) of tuples \( (i, j) \) where \( 0 \leq i < M, 0 \leq j < N \). A particular pair of values \( i, j \) can only appear at most once in \( M \). We define \( \text{nrows}(M) = N \), and \( \text{ncols}(M) = M \).

We also define the sets \( \text{indrow}(M) = \{ i : \exists (i, j) \in L(M) \} \) and \( \text{indcol}(M) = \{ j : \exists (i, j) \in L(M) \} \).

These are the sets of nonempty rows and columns of \( M \), respectively. The structure of a two-dimensional mask \( M \) is the set \( \text{ind}(M) = \{ (i, j) : (i, j) \in L(M) \} \).

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

It is the set of all possible indices that do not appear in \( M \).

### 3.7 Descriptors

Descriptors, the last argument in all GraphBLAS methods, are used to provide more details for the operation to be performed by those methods. In particular, descriptors specify how the other input arguments that correspond to collections – vectors, matrices and masks – should be processed (modified) before the main operation of a method is performed.

The descriptor is a lightweight object. It pairs a set of flags representing the possible modifiers with each collection argument of the GraphBLAS method. For example, a descriptor may specify that a particular input matrix needs to be transposed or that a mask needs to be structurally complemented (defined in Section 3.6) before using it in the operation.
For the purpose of constructing descriptors, the arguments of a method that can be modified are identified by specific field names. The output parameter (typically the first parameter in a GraphBLAS method) is indicated by the field name, `GrB.OUTP`. The mask is indicated `GrB.MASK` field name. The input parameters corresponding to the input vectors and matrices are indicated by `GrB.INP0` and `GrB.INP1`, in the order they appear in the signature of the GraphBLAS method.
Chapter 4

Methods

All methods can be declared for use in programs by including the GraphBLAS.h header file.

4.1 Context Methods

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

4.1.1 init: Initialize a GraphBLAS context

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

- GrB_Blocking: Methods in a sequence return after computations in the method have completed and output arguments are available to subsequent statements in an application. When executing in GrB_Blocking mode, the methods execute in program order.

- GrB_NonBlocking: Methods in a sequence return after arguments in the method have been tested for consistency with the method but potentially before computations complete or output arguments are available to subsequent statements in an application. When executing in GrB_NonBlocking mode, the methods in a sequence may execute in any order that preserves the mathematically result defined by the sequence.

GrB_init() may be called with GrB_NULL to select the default mode. This mode is implementation defined.

C Syntax

GrB_info GrB_init(GrB_Mode m);
Parameters

\( m \) Mode for the GraphBLAS context.

Return Values

\text{GrB\_SUCCESS} operation completed successfully

\text{GrB\_PANIC} unknown internal error

\text{GrB\_NOMODE} mode does not exist

4.1.2 finalize: Finalize a GraphBLAS context

Terminates and frees any internal resources created to support the GraphBLAS C API context. An application may not create a new context after \text{GrB\_finalize} has been called.

C Syntax

\begin{verbatim}
GrB_info GrB_finalize();
\end{verbatim}

Return Values

\text{GrB\_SUCCESS} operation completed successfully

\text{GrB\_PANIC} unknown internal error

4.2 Object Methods

4.2.1 Algebra Methods

4.2.1.1 Type\_new: Create a new user-defined type

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

C Syntax

\begin{verbatim}
GrB_info GrB_Type_new(GrB_Type *utype,
                        <in_type> type);
\end{verbatim}
Parameters

utype Pointer to the GrB_Type object that will be initialized to the new user-defined type.

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

Return Values

GrB_SUCCESS operation completed successfully.

GrB_PANIC unknown internal error.

GrB_OUTOFMEM not enough memory available for operation.

Description

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

\[
\text{memcpy}(&\text{dest}, &\text{src}, \text{sizeof(type)})
\]

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

4.2.1.2 UnaryOp_new: Create a new unary operator

Initializes a new GraphBLAS unary operation with specified domains and user-defined function.

C Syntax

\[
\text{GrB_info GrB_UnaryOp_new(GrB_UnaryOp *} *\text{unary}_\text{op,}
\]

\[
\text{GrB_Type } \text{d1,}
\]

\[
\text{GrB_Type } \text{d2,}
\]

\[
\text{void } *\text{unary}_\text{func);}
\]

Parameters

unary_op Identifier of the newly created UnaryOp.

d1 The GrB_Type defining the domain of input argument of the unary function. Should be one of the predefined GraphBLAS types in Table 2.1 or a user created type.
The \texttt{GrB} \texttt{Type} defining the output domain of the function being created. Should be one of the predefined GraphBLAS types in Table \ref{tab:graphblas-types} or a user created type.

\texttt{unary\_func} a pointer to a user-defined function that takes one input parameter of a type consistent with \texttt{d1}’s domain and returns a value of type consistent with \texttt{d2}’s domain.

\section*{Return Values}

\begin{itemize}
  \item \texttt{GrB\_SUCCESS} operation completed successfully.
  \item \texttt{GrB\_PANIC} unknown internal error.
  \item \texttt{GrB\_OUTOFMEM} not enough memory available for operation.
  \item \texttt{GrB\_DOMAIN\_MISMATCH} the types in the function pointer signature are not consistent with the \texttt{GrB\_Type} parameters specified.
\end{itemize}

\section*{Description}

Creates a new GraphBLAS unary operator \( f = (\text{D}(\texttt{d1}), \text{D}(\texttt{d2}), \text{unary\_func}) \) and returns its identifier in \texttt{unary\_op}.

\subsection*{4.2.1.3 BinaryOp\_new: Create a new binary operator}

Initializes a new GraphBLAS binary operator with specified domains and user-defined function.

\section*{C Syntax}

\begin{verbatim}
GrB\_info GrB\_BinaryOp\_new(GrB\_BinaryOp *binary\_op,
  GrB\_Type d1,
  GrB\_Type d2,
  GrB\_Type d3,
  void *binary\_func);
\end{verbatim}

\section*{Parameters}

\begin{itemize}
  \item \texttt{binary\_op} Identifier of the newly created BinaryOp.
  \item \texttt{d1} The \texttt{GrB\_Type} defining the domain of left hand argument of the binary function. Should be one of the predefined GraphBLAS types in Table \ref{tab:graphblas-types} or a user created type.
  \item \texttt{d2} The \texttt{GrB\_Type} defining the domain of the right hand argument of the binary function. Should be one of the predefined GraphBLAS types in Table \ref{tab:graphblas-types} or a user created type.
\end{itemize}
d3 The `GrB_Type` defining the output domain of the binary function. Should be one of the predefined GraphBLAS types in Table 2.1 or a user created type.

binary_func A pointer to a user-defined function that takes two input parameters of types consistent with d1 and d2 and returns a value with the type consistent with d3.

Return Values

- `GrB_SUCCESS` operation completed successfully.
- `GrB_PANIC` unknown internal error.
- `GrB_OUTOFMEM` not enough memory available for operation.
- `GrB_DOMAIN_MISMATCH` the types in the function pointer signature are not consistent with the `GrB_Type` parameters specified.

Description

Creates a new GraphBLAS binary operator \( f = \langle D(d1), D(d2), D(d3), \text{binary_func} \rangle \) and returns its identifier in `binary_op`.

4.2.1.4 Monoid_new: Create new monoid

Creates a new monoid with specified domain, operator, and identity element.

C Syntax

```c
GrB_info GrB_Monoid_new(GrB_Monoid *monoid,
GrB_Type d1,
GrB_BinaryOp binary_op,
<d1_type> identity);
```

Parameters

- `monoid` Identifier of the newly created monoid.

- d1 The `GrB_Type` defining the domain of the monoid being created. It should be one of the predefined GraphBLAS types in Table 2.1 or a user created type.

- binary_op A GraphBLAS binary operator with input types consistent with d1, and returns a value of type d1.

- identity The identity element of the monoid. Must be of type corresponding to the value of d1 according to Table 2.1.
Return Values

GrB_SUCCESS operation completed successfully.

GrB.PANIC unknown internal error.

GrB.OUTOFMEM not enough memory available for this method to complete.

GrB.DOMAIN.MISMATCH the types in the operator signature are not consistent with the GrB.Type parameter specified.

Description

Creates a new monoid \( M = \langle D(d_1), \text{binary}_\text{op}, \text{identity} \rangle \) and returns its identifier in monoid.

4.2.1.5 Semiring_new: Create new semiring

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

C Syntax

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

Parameters

semiring Identifier of the newly created semiring.

add_op A conventional monoid that specifies the addition operator and its identity.

mul_op A binary operator that specifies the semiring’s multiplication.

Return Values

GrB.SUCCESS operation completed successfully.

GrB.PANIC unknown internal error.

GrB.OUTOFMEM not enough memory available for this method to complete.

GrB.DOMAIN.MISMATCH the output domain of mul_op does not match the domain of the add_op.
Description

Creates a new semiring \( S = \langle D_1(\text{mul\_op}), D_2(\text{mul\_op}), D(\text{add\_op}), \text{add\_op}, \text{mul\_op}, 0(\text{add\_op}) \rangle \) and returns its identifier in semiring. Note that \( D_3(\text{mul\_op}) \) must be the same as \( D(\text{add\_op}) \).

4.2.2 Vector Methods

4.2.2.1 Vector_new: Create new vector

Creates a new vector with specified domain and size.

C Syntax

```c
GrB_info GrB_Vector_new(GrB_Vector *v,
                        GrB_Type d,
                        GrB_Index n);
```

Parameters

- \( v \) Identifier of the newly created vector.
- \( d \) The type defining the domain (scalar type) of the vector being created. Should be one of the predefined GraphBLAS types in Table 2.1 or a user created type.
- \( n \) The size of the vector being created.

Return Values

- \( \text{GrB\_SUCCESS} \) operation completed successfully.
- \( \text{GrB\_PANIC} \) unknown internal error.
- \( \text{GrB\_OUTOFMEM} \) not enough memory available for this method to complete.

Description

Creates a new vector \( v \) of domain \( D(d) \), size \( n \), and empty \( L(v) \). It returns in \( v \) this vector \( v \).

4.2.2.2 Vector_clear: Clear a vector

Removes all the elements from a vector.
GrB_info GrB_Vector_clear(GrB_Vector *v);

Parameters

v Identifier of the vector to clear.

Return Values

GrB_SUCCESS operation completed successfully.
GrB_PANIC unknown internal error.
GrB_NOVECTOR vector does not exist.

Description

Removes all tuples from an existing vector.

4.2.2.3 Vector_size: Size of a vector

Retrieve the size of a vector.

GrB_info GrB_Vector_size(GrB_Index *n,
                        const GrB_Vector v);

Parameters

n On successful return, is set to the size \(N\) of the vector.

v Vector being queried.

Return Values

GrB_SUCCESS operation completed successfully.
GrB_PANIC unknown internal error.
GrB_NOVECTOR vector does not exist.
Description

Return in \( n \) the size (parameter \( N \) in Section \[3.4\]) in vector \( v \).

4.2.2.4 Vector\_nnz: Number of stored elements in a vector

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

C Syntax

```c
GrB_info GrB_Vector_nnz(GrB_Index *nnz, 
   const GrB_Vector v);
```

Parameters

- \( \text{nnz} \) On successful return, is set to the number of stored elements (tuples) in the vector.
- \( v \) Vector being queried.

Return Values

- \( \text{GrB\_SUCCESS} \) operation completed successfully.
- \( \text{GrB\_PANIC} \) unknown internal error.
- \( \text{GrB\_NOVECTOR} \) vector does not exist.

Description

Return in \( \text{nnz} \) the number of stored elements (the size of \( L(v) \) in Section \[3.4\]) in vector \( v \).

4.2.3 Matrix Methods

4.2.3.1 Matrix\_new: Create new matrix

Creates a new matrix with specified domain and dimensions.

C Syntax

```c
GrB_info GrB_Matrix_new(GrB_Matrix *A, 
   GrB_Type d, 
   GrB_Index m, 
   GrB_Index n);
```
Parameters

A Identifier of the newly created matrix.

d The type defining the domain of the matrix being created. Should be one of the predefined GraphBLAS types in Table 2.1 or a user created type.

m The number of rows of the matrix being created.

n The number of columns of the matrix being created.

Return Values

GrB_SUCCESS operation completed successfully.

GrB_PANIC unknown internal error.

GrB_OUTOFMEM not enough memory available for this method to complete.

Description

 Creates a new matrix A of domain D(d), size m x n, and empty L(A). It returns in A this matrix A.

4.2.3.2 Matrix_clear: Clear a matrix

Removes all elements from a matrix.

C Syntax

GrB_info GrB_Matrix_clear(GrB_Matrix A);

Parameters

A Identifier of the matrix to clear.

Return Values

GrB_SUCCESS operation completed successfully.

GrB_PANIC unknown internal error.

GrB_NOMATRIX matrix does not exist.
Description

Removes all elements (tuples) from an existing matrix.

4.2.3.3 Matrix_nrows: Number of rows in a matrix

Retrieve the number of rows in a matrix.

C Syntax

```c
GrB_info GrB_Matrix_nrows(GrB_Index *m,
const GrB_Matrix A);
```

Parameters

- `m` On successful return, contains the number of rows in the matrix.
- `A` Matrix being queried.

Return Values

- `GrB_SUCCESS` operation completed successfully.
- `GrB_PANIC` unknown internal error.
- `GrB_NOMATRIX` matrix does not exist.

Description

Return in `m` the number of rows (parameter `M` in Section 3.5) in matrix `A`.

4.2.3.4 Matrix_ncols: Number of columns in a matrix

Retrieve the number of columns in a matrix.

C Syntax

```c
GrB_info GrB_Matrix_ncols(GrB_Index *n,
const GrB_Matrix A);
```
Parameters

- $n$ On successful return, contains the number of columns in the matrix.
- $A$ Matrix being queried.

Return Values

- GrB_SUCCESS operation completed successfully.
- GrB_PANIC unknown internal error.
- GrB_NOMATRIX matrix does not exist.

Description

Return in $n$ the number of columns (parameter $N$ in Section 3.5) in matrix $A$.

### 4.2.3.5 Matrix$_{\text{nnz}}$: Number of stored elements in a matrix

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

C Syntax

```c
GrB_info GrB_Matrix_nnz(GrB_Index *nnz,
                          const GrB_Matrix A);
```

Parameters

- $nnz$ On successful return, contains the number of stored elements (tuples) in the matrix.
- $A$ Matrix being queried.

Return Values

- GrB_SUCCESS operation completed successfully.
- GrB_PANIC unknown internal error.
- GrB_NOMATRIX matrix does not exist.

Description

Return in $nnz$ the number of tuples (the size of $L(A)$ in Section 3.5) stored in matrix $A$. 

36
4.2.4 Descriptor Methods

4.2.4.1 Descriptor_new: Create new descriptor

Creates a new (empty) descriptor.

C Syntax

GrB_info GrB_Descriptor_new(GrB_Descriptor *desc);

Parameters

desc Identifier of new descriptor created.

Return Value

GrB_SUCCESS operation completed successfully.
GrB_PANIC unknown internal error.
GrB_OUTOFMEM not enough memory available for this method to complete.

Description

Returns in desc the identifier of a newly created empty descriptor. A newly created descriptor can be populated with calls to Descriptor_set.

4.2.4.2 Descriptor_set: Set content of descriptor

Sets the content (details of an operation) for a field of an existing descriptor.

C Syntax

GrB_info GrB_Descriptor_set(GrB_Descriptor desc, 
GrB_Field field, 
GrB_Value val);

Parameters

desc The descriptor being modified by this method.
field The descriptor field being set.
val New value for the field being set.

Return Values

GrB_SUCCESS operation completed successfully.

GrB_PANIC unknown internal error.

GrB_OUTOFMEM not enough memory available for operation.

GrB_INVALID_VALUE invalid value set on the field.

Description

Valid values for the field parameter include the following:

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

GrB_MASK refers to the mask parameter of the operation.

GrB_INP_{x} refers to the input parameters of the operation (matrices and vectors), where

\(x\) is '0' for the first input argument, '1' is for the second, and so on.

Valid values for the val parameter are built from one or more of the following together:

GrB_SCMP Use the structural complement of the corresponding mask (GrB_MASK) parameter

GrB_TRAN compute the transpose of the corresponding parameter (valid for input ma-

trice parameters only).

When multiple modifiers need to be specified for a given field, the value parameters should be

OR-ed together and a single call to set is used.

4.2.5 free: Destroy object

Destroys a previously created GraphBLAS object and releases any resources used.

C Syntax

GrB_info GrB_free(GrB_Object obj);
Parameters

obj  GraphBLAS object to be destroyed. Can be a matrix, vector or descriptor.

Return Values

GrB_SUCCESS  operation completed successfully

GrB_PANIC  unknown internal error

GrB_NOOBJECT  object does not exist

Description

4.3 GraphBLAS Operations

Table 4.1: A Mathematical overview of the fundamental GraphBLAS operations supported in this specification. Input matrices $A$ and $B$ may be optionally transposed. Use of an optional mask is indicated, for example when applied to the matrix $C$, as $C(M)$. The mask or its structural compliment controls which values are written into the output result.

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Mathematical Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxm</td>
<td>$C(M) \oplus = A \odot B$</td>
</tr>
<tr>
<td>mxv</td>
<td>$u(m) \oplus = A \odot v$</td>
</tr>
<tr>
<td>vxm</td>
<td>$u(m) \oplus = v \odot A$</td>
</tr>
<tr>
<td>eWiseMult</td>
<td>$C(M) \oplus = A \otimes B$</td>
</tr>
<tr>
<td>eWiseAdd</td>
<td>$C(M) \oplus = A \oplus B$</td>
</tr>
<tr>
<td>reduce (row)</td>
<td>$u(m) \oplus = \oplus_j A(:,j)$</td>
</tr>
<tr>
<td>apply</td>
<td>$C(M) \oplus = f(A)$</td>
</tr>
<tr>
<td>transpose</td>
<td>$C(M) \oplus = A$</td>
</tr>
<tr>
<td>extract</td>
<td>$C(M) \oplus = A(i,j)$</td>
</tr>
<tr>
<td>assign</td>
<td>$C(M)(i,j) \oplus = A$</td>
</tr>
<tr>
<td>buildMatrix</td>
<td>$C(M) \oplus = S^{m \times n}(i,j,v,\oplus_{dup})$</td>
</tr>
<tr>
<td>buildVector</td>
<td>$u(m) \oplus = S^n(i,v)$</td>
</tr>
<tr>
<td>extractTuples</td>
<td>$(i,j,v) = A(M)$</td>
</tr>
</tbody>
</table>

The GraphBLAS operations are defined in the GraphBLAS math spec and summarized in Table 4.1. In addition to methods that implement these fundamental GraphBLAS operations, we support a number of variants that have been found to be especially useful in algorithm development. A GraphBLAS operation is only valid when the domains of the GraphBLAS objects are mathematically consistent. The C programming language defines implicit casts between data types. For example, floats, doubles and ints can be freely mixed according to the rules defined for implicit casts. It is the responsibility of the user to assure that these casts are appropriate for the algorithm.
in question. For example, a cast to int implies truncation of a floating point type. Depending on
the operation, this truncation error could lead to erroneous results. Furthermore, casting a wider
type onto a narrower type can lead to overflow errors. The GraphBLAS operations do not attempt
to protect a user from these sorts of errors.

GraphBLAS operations also make assumptions about the numbers of dimensions and sizes of objects
in an operation. An operation will test these sizes and report an error if they are inconsistent. For
example, when multiplying two matrices, \( C = A \times B \), the number of rows of \( C \) must equal the number
of rows of \( A \), the number of columns of \( A \) must match the number of rows of \( B \), and the number of
columns of \( C \) must match the number of columns of \( B \). This is the behavior expected given the
mathematical definition of the operations.

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

The methods that implement the GraphBLAS operations test the domains and the sizes in each
dimension based on the input arguments and the descriptors. Error conditions are reported when
mismatches occur.

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

Given a GraphBLAS vector \( v = \langle D, N, \{(i, v_i)\} \rangle \), a one-dimensional mask \( m = \langle N, \{i : (\text{bool}) v_i = \text{true}\} \rangle \) is derived for use in the operation, where \((\text{bool}) v_i\) denotes casting the value \( v_i \) to a Boolean
value (true or false). We note that, if cast is disallowed for the mask by the operation descriptor,
then \( D(v) \) must be \( \text{GrB BOOL} \).

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

In both the one- and two-dimensional cases, the mask may go through a structural complement
operation (§ 3.6) as specified in the descriptor, before a final mask is generated for use in the
operation.
4.3.1 buildMatrix: Store elements from tuples into a matrix

C Syntax

```c
GrB_info GrB_buildMatrix(GrB_Matrix *C,
    const GrB_Matrix Mask,
    const GrB_BinaryOp accum,
    const GrB_Index *rowIDs,
    const GrB_Index *colIDs,
    const <type> *values,
    GrB_Index n,
    const GrB_BinaryOp dup,
    const GrB_Descriptor desc);
```

Parameters

- **C** (GrB_OUTP) An existing Matrix object to store the result.
- **Mask** (GrB_MASK) Output mask specifies which locations in C can be modified. If no mask is desired, GrB_NULL should be specified.
- **accum** Operator used for accumulating entries into existing C entries. If no accumulation is desired, GrB_NULL should be specified.
- **rowIDs** Pointer to an array of row indices.
- **colIDs** Pointer to an array of column indices.
- **values** (GrB_INP0) Pointer to an array of scalars of a type that is compatible with the domain of matrix, C.
- **n** The number of values contained in each array.
- **dup** A binary function to apply when duplicate values for the same location are present in the input arrays.
- **desc** Operation descriptor. If a default descriptor is desired, GrB_NULL can be used. Valid fields and values are as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_MASK</td>
<td>GrB_SCMP</td>
<td>Use the structural complement of the mask.</td>
</tr>
</tbody>
</table>

Return Values

- GrB_SUCCESS operation completed successfully.
- GrB_PANIC unknown internal error.
GrB_OUTOFMEM not enough memory available for operation.

GrB_NOMATRIX C does not existing

GrB_INDEX_OUTOFBOUNDS A value in i references a nonexistent row in C, or the value in j references a nonexistent column in C (matrix version).

GrB_DOMAIN_MISMATCH mismatch between value type and matrix domain.

Description

Each tuple \{\text{rowIDs}[i], \text{colIDs}[i], \text{values}[i]\} is a contribution to the output in the form of

\[ C[\text{rowIDs}[i], \text{colIDs}[i]] = \text{accum}(\text{values}[i], C[\text{rowIDs}[i], \text{colIDs}[i]]) . \]

If \text{accum} parameter is not provided, then the contribution is of the form

\[ C[\text{rowIDs}[i], \text{colIDs}[i]] = \text{values}[i] . \]

If multiple values for the same location are present in the input arrays, the \text{dup} function is used to reduce them before assignment or accumulation into \( C \).

If a mask is used, then only locations in \( C \) specified by the mask can be assigned or accumulated into.

\text{rowIDs}, \text{colIDs}, \text{and} \text{values} should be of the same length.

4.3.2 buildVector: Store elements from tuples into a vector

C Syntax

```c
GrB_info GrB_buildVector(GrB_Vector *u,
const GrB_Vector mask,
const GrB_BinaryOp accum,
const GrB_Index *indices,
const <type> *values,
GrB_Index n,
const GrB_BinaryOp dup,
const GrB_Descriptor desc);
```

Parameters

\( u \) (GrB_OUTP) An existing Vector object to store the result.

\( \text{mask} \) (GrB_MASK) Output mask specifies which locations in \( u \) can be modified. If no mask is desired, \( \text{GrB_NULL} \) should be specified.
accum Operator used for accumulating entries into existing u entries. If no accumulation is desired, GrB.NULL should be specified.

indices Pointer to an array of indices.

values (GrB_INP0) Pointer to an array of scalars of a type that is compatible with the domain of vector u.

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

dup A binary function to apply when duplicate values for the same location are present in the input arrays.

desc Operation descriptor. If a default descriptor is desired, GrB.NULL can be used. Valid fields and values are as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_MASK</td>
<td>GrB_SCMP</td>
<td>Use the structural complement of the mask.</td>
</tr>
</tbody>
</table>

Return Values

GrB_SUCCESS Operation completed successfully.

GrB_PANIC Unknown internal error.

GrB_OUTOFMEM Not enough memory available for operation.

GrB_NOVECTOR u does not exist

GrB_INDEX_OUTOFBOUNDS A value in indices is outside the allowed range for u.

GrB_DOMAIN_MISMATCH Mismatch between value type and vector domain, or mask domain and bool.

Description

Compute mask vector m from input parameter mask based on descriptor desc. Then, for i = 0, . . . , n − 1, do the following:

1. If indices[i] \notin L(m), continue to next i.

2. If indices[i] \notin i(u), then L(u) ← L(u) \cup (indices[i], values[i]).

3. If indices[i] \in i(u), then replace the tuple (indices[i], v_{indices[i]}) \in L(u) with the tuple (indices[i], accum(v_{indices[i]}, values[i])).

After a call to GrB_buildVector, the program should perform a GrB_wait on vector u before modifying or deleting arrays indices and values.
### 4.3.3 `extractTuples`: Extract tuples from a matrix

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

#### C Syntax

```c
GrB_info GrB_extractTuples(GrB_Index *rowIDs,
                         GrB_Index *colIDs,
                         <type> *values,
                         const GrB_Matrix A,
                         const GrB_Matrix Mask,
                         const GrB_Descriptor desc,
                         char *err);
```

#### Parameters

- **rowIDs** Pointer to an array of row indices that is sufficient to hold all of the row indices (no checking is performed).
- **colIDs** Pointer to an array of column indices that is sufficient to hold all of the column indices (no checking is performed).
- **values** Pointer to an array of scalars of a type that is sufficient to hold all of the stored values (no checking is performed) whose type is compatible with \(D(A)\).
- **A** (GrB_INP0) An existing GraphBLAS matrix.
- **Mask** (GrB_MASK) Input mask specifies which locations in \(A\) that can be extracted. If no mask is desired, \(GrB_NULL\) should be specified.
- **desc** Operation descriptor. If a default descriptor is desired, \(GrB_NULL\) is to be used. Valid fields and values are as follows:
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_MASK</td>
<td>GrB_SCMP</td>
<td>Use the structural complement of the mask.</td>
</tr>
</tbody>
</table>
- **err** A null terminated string containing additional error information.

#### Return Values

- **GrB_SUCCESS** Operation completed successfully.
- **GrB_PANIC** Unknown internal error.
- **GrB_NOMATRIX** \(A\) does not exist
- **GrB_INDEX_OUTOFBOUNDS** A value in \(indices\) is outside the allowed range for \(A\).
- **GrB_DIMENSION_MISMATCH** Mismatch between dimensions of matrix and mask.
GrB_DOMAIN_MISMATCH Mismatch between value type and matrix domain, or mask domain and bool.

Description

4.3.4 extractTuples: Extract tuples from a vector

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

C Syntax

```c
GrB_info GrB_extractTuples(GrB_Index *indices,
                          <type> *values,
                          const GrB_Vector v,
                          const GrB_Vector mask,
                          const GrB_Descriptor desc,
                          char *err);
```

indices Pointer to an array of indices that is sufficient to hold all of the stored values’ indices (no checking is performed).

values Pointer to an array of scalars of a type that is sufficient to hold all of the stored values (no checking is performed) whose type is compatible with $D(v)$.

v (GrB_INPO) An existing GraphBLAS vector.

mask (GrB_MASK) Input mask specifies which locations in $v$ that can be extracted. If no mask is desired, GrB_NULL should be specified.

desc Operation descriptor. If a default descriptor is desired, GrB_NULL is to be used. Valid fields and values are as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_MASK</td>
<td>GrB_SCMP</td>
<td>Use the structural complement of the mask, $\overline{M}$.</td>
</tr>
</tbody>
</table>

err A null terminated string containing additional error information.

Return Values

- **GrB_SUCCESS** Operation completed successfully.
- **GrB_PANIC** Unknown internal error.
- **GrB_NOVECTOR** $v$ does not exist
- **GrB_INDEX_OUTOFBOUNDS** A value in indices is outside the allowed range for $u$. 

45
GrB_DIMENSION_MISMATCH  Mismatch between dimensions of vector and mask.
GrB_DOMAIN_MISMATCH  Mismatch between value type and matrix domain, or mask domain and bool.

Description
4.3.5 mxm: Matrix-matrix multiply

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

C Syntax

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

Parameters

- **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the matrix product. On output, the matrix holds the results of this operation.
- **Mask** (IN) A “write” mask that controls which results from this operation are stored into the output matrix C (optional). If no mask is desired (i.e., all elements of result are copied into the output matrix), GrB_NULL should be specified. The Mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any of the predefined “built-in” types in Table 2.1.
- **accum** (IN) Operator used for accumulating entries into existing C entries: \(\langle D_x, D_y, D_z, \odot \rangle\). If assignment rather than accumulation is desired, GrB_NULL should be specified.
- **op** (IN) Semiring used in the matrix-matrix multiply: \(\langle D_1, D_2, D_3, \oplus, \otimes, 0 \rangle\).
- **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the multiplication.
- **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the multiplication.
- **desc** (IN) Operation descriptor (optional). If a default descriptor is desired, GrB_NULL should be used. Valid fields are as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GrB_OUTP</td>
<td>GrB_REPLACE</td>
<td>Output matrix C is cleared (all elements removed) before result is stored in it.</td>
</tr>
<tr>
<td>Mask</td>
<td>GrB_MASK</td>
<td>GrB_SCMP</td>
<td>Use the structural complement of Mask.</td>
</tr>
<tr>
<td>A</td>
<td>GrB_INP0</td>
<td>GrB_TRAN</td>
<td>Use transpose of A for operation.</td>
</tr>
<tr>
<td>B</td>
<td>GrB_INP1</td>
<td>GrB_TRAN</td>
<td>Use transpose of B for operation.</td>
</tr>
</tbody>
</table>
Return Values

GrB_SUCCESS  In blocking mode, operation completed successfully. In non-blocking mode, this indicates that the consistency 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_OUTOFMEM  Not enough memory available for operation

GrB_DIMENSION_MISMATCH  Matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH  The domains of the various matrices are incompatible with the corresponding domains of the accumulating operation, semiring, or mask.

Description

GrB_mxm computes the matrix product C = A ⊗ ⊕ B or, if an optional binary accumulation operator (⊙) is provided, C = C ⊙ A ⊗ ⊕ B. (Matrices A and B can be optionally transposed.) Logically, this operation occurs in three steps:

1. The internal matrices and mask used in the computation are formed and their domains/dimensions are tested for consistency.
2. The indicated computations are carried out.
3. The result is written into the output matrix, possibly under control of a mask.

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

1. C = ⟨D(C), nrows(C), ncols(C), L(C) = {(i, j, C_{ij})}⟩
2. Mask = ⟨D(Mask), nrows(Mask), ncols(Mask), L(Mask) = {(i, j, Mask_{ij})}⟩ (optional)
3. A = ⟨D(A), nrows(A), ncols(A), L(A) = {(i, j, A_{ij})}⟩
4. B = ⟨D(B), nrows(B), ncols(B), L(B) = {(i, j, B_{ij})}⟩

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

1. The domain of Mask (if not GrB_NULL) must be from one of the pre-defined types of Table 2.1.
2. D(A) must be compatible with D₁ of the semiring.
3. D(B) must be compatible with D₂ of the semiring.
4. If \texttt{accum} is \texttt{GrB\_NULL}, then \texttt{D(C)} must be compatible with \texttt{D}_3 of the semiring.

5. If \texttt{accum} is not \texttt{GrB\_NULL}, then \texttt{D(C)} must be compatible with \texttt{D}_x and \texttt{D}_z of the accumulator operator and \texttt{D}_3 of the semiring must be compatible with \texttt{D}_y of the accumulator operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 2.1 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any consistency rule above is violated, execution of \texttt{GrB\_mxm} ends and the domain mismatch error listed above is returned.

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

1. Matrix \(\tilde{C} \leftarrow C\).

2. Two-dimensional mask \(\tilde{\text{Mask}}\) is computed from argument \(\text{Mask}\) as follows:
   
   (a) If \(\text{Mask} = \text{GrB\_NULL}\), then \(\tilde{\text{Mask}} = \langle \text{nrows}(C), \text{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \text{nrows}(C), 0 \leq j < \text{ncols}(C)\}\rangle\).
   
   (b) Otherwise, \(\tilde{\text{Mask}} = \langle \text{nrows}(\text{Mask}), \text{ncols}(\text{Mask}), \{(i, j) : (\text{bool})\text{Mask}(i, j) = \text{true}\}\rangle\).
   
   (c) If \(\text{desc}[\text{GrB\_MASK}], \text{GrB\_SCMP}\) is true, then \(\tilde{\text{Mask}} \leftarrow \neg \tilde{\text{Mask}}\).

3. Matrix \(\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}], \text{GrB\_TRAN} \ ? A^T : A\).

4. Matrix \(\tilde{B} \leftarrow \text{desc}[\text{GrB\_INP1}], \text{GrB\_TRAN} \ ? B^T : B\).

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

1. \(\text{nrows}(\tilde{C}) = \text{nrows}(\tilde{\text{Mask}})\).
2. \(\text{ncols}(\tilde{C}) = \text{ncols}(\tilde{\text{Mask}})\).
3. \(\text{nrows}(\tilde{C}) = \text{nrows}(\tilde{A})\).
4. \(\text{ncols}(\tilde{C}) = \text{ncols}(\tilde{B})\).
5. \(\text{ncols}(\tilde{A}) = \text{nrows}(\tilde{B})\).

If any consistency rule above is violated, execution of \texttt{GrB\_mxm} ends and the dimension mismatch error listed above is returned.

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

- \(\tilde{T}\): The matrix holding the product of matrices \(\tilde{A}\) and \(\tilde{B}\).
- \(\tilde{Z}\): The matrix holding the result after application of the (optional) accumulator.
The intermediate matrix $\tilde{T} = \langle D_3, nrows(\tilde{A}), ncols(\tilde{B}), L(T) = \{(i, j, T_{ij}) : \text{ind}(\tilde{A}(i,:)) \cap \text{ind}(\tilde{B}(j,:)) \neq \emptyset\}\rangle$ is created. The value of each of its elements is computed by

$$T_{ij} = \bigoplus_{k \in \text{ind}(\tilde{A}(i,:)) \cap \text{ind}(\tilde{B}(j,:))} (\tilde{A}(i,k) \odot \tilde{B}(k,j)),$$

where $\oplus$ and $\odot$ are the additive and multiplicative operators of semiring $\text{op}$, respectively.

The intermediate matrix $\tilde{Z}$ is created as follows: If $\text{accum} = \text{GrB\_NULL}$, then $\tilde{Z} = \tilde{T}$. If $\text{accum} = \langle D_x, D_y, D_z, \odot \rangle$, then matrix $\tilde{Z}$ is defined as $\langle D_z, nrows(\tilde{C}), ncols(\tilde{C}), L(\tilde{Z}) = \{(i, j, Z_{ij}) : (i, j) \in \text{ind}(\tilde{C}) \cap \text{ind}(\tilde{T})\}\rangle$. The values of the elements of $\tilde{Z}$ are computed based on the relationships between the sets of indices in $\tilde{C}$ and $\tilde{T}$.

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

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

Finally, the set of output values that make up the $\tilde{Z}$ matrix are written into the final result matrix, $\tilde{C}$. This is carried out under control of the mask which acts as a “write mask”. If $\text{desc[GrB\_OUTP].GrB\_REPLACE}$ is true, then any values in $\tilde{C}$ on input to $\text{GrB\_mxm()}$ are deleted and the new output matrix $\tilde{C}$ is,

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

If $\text{desc[GrB\_OUTP].GrB\_REPLACE}$ is not given or is equal to any value other than true, the elements of $\tilde{Z}$ indicated by the mask are copied into the result matrix, $\tilde{C}$ and elements of $\tilde{C}$ that fall outside the set indicated by the mask are unchanged:

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

In $\text{GrB\_Blocking}$ mode, the method exits with return value $\text{GrB\_SUCCESS}$ and the new content of matrix $\tilde{C}$ is as defined above and fully computed. In $\text{GrB\_NonBlocking}$ mode, the method exits with return value $\text{GrB\_SUCCESS}$ and the new content of matrix $\tilde{C}$ is as defined above but may not be fully computed. (It can be used in the next GraphBLAS method call in a sequence.)
4.4 Sequence Termination

4.4.1 wait: Waits until pending operations complete

When running in non-blocking mode, this function guarantees that all pending GraphBLAS operations are fully executed. Note that this can be called in blocking mode without an error, but there should be no pending GraphBLAS operations to complete.

C Syntax

```c
GrB_info GrB_wait(char* err)
```

Parameters

- `err` A null terminated string containing additional error information.

Return values

- `GrB_SUCCESS` operation completed successfully
- `GrB_PANIC` unknown internal error; more information about the error may be found in the `err` string.

Description

Upon successful return, all previously called GraphBLAS methods have fully completed their execution, and any (transparent or opaque) data structures produced or manipulated by those methods can be safely touched. If an error occurred in any pending GraphBLAS operations, `GrB_PANIC` is returned and the `err` string will contain implementation defined error information about the problem encountered.