User Guide for SuiteSparse:GraphBLAS

Timothy A. Davis
davis@tamu.edu, Texas A&M University.
http://suitesparse.com
https://people.engr.tamu.edu/davis
https://twitter.com/DocSparse

VERSION 6.2.0, Feb 14, 2022

Abstract

SuiteSparse:GraphBLAS is a full implementation of the GraphBLAS standard, which defines a set of sparse matrix operations on an extended algebra of semirings using an almost unlimited variety of operators and types. When applied to sparse adjacency matrices, these algebraic operations are equivalent to computations on graphs. GraphBLAS provides a powerful and expressive framework for creating high-performance graph algorithms based on the elegant mathematics of sparse matrix operations on a semiring.

When compared with MATLAB R2021a, some methods in GraphBLAS are up to a million times faster than MATLAB, even when using the same syntax. Typical speedups are in the range 2x to 30x. The statement `C(M)=A` when using MATLAB sparse matrices takes $O(e^2)$ time where e is the number of entries in `C`. GraphBLAS can perform the same computation with the exact same syntax, but in $O(e \log e)$ time (or $O(e)$ in some cases), and in practice that means GraphBLAS can compute `C(M)=A` for a large problem in under a second, while MATLAB takes about 4 to 5 days.

Contents

1 Introduction

The GraphBLAS standard defines sparse matrix and vector operations on an extended algebra of semirings. The operations are useful for creating a wide range of graph algorithms.

For example, consider the matrix-matrix multiplication, $\mathbf{C} = \mathbf{AB}$. Suppose $\mathbf{A}$ and $\mathbf{B}$ are sparse n-by-n Boolean adjacency matrices of two undirected graphs. If the matrix multiplication is redefined to use logical AND instead of scalar multiply, and if it uses the logical OR instead of add, then the matrix $\mathbf{C}$ is the sparse Boolean adjacency matrix of a graph that has an edge (i, j) if node i in $\mathbf{A}$ and node j in $\mathbf{B}$ share any neighbor in common. The OR-AND pair forms an algebraic semiring, and many graph operations like this one can be succinctly represented by matrix operations with different semirings and different numerical types. GraphBLAS provides a wide range of built-in types and operators, and allows the user application to create new types and operators without needing to recompile the GraphBLAS library.

For more details on SuiteSparse:GraphBLAS, and its use in LAGraph, see [Dav19, Dav21, Dav18, DAK19, ACD+20, MDK+19].

A full and precise definition of the GraphBLAS specification is provided in *The GraphBLAS C API Specification* by Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira, Carl Yang, and Benjamin Brock [BMM+17a, BMM+17b, BBM+21], based on *GraphBLAS Mathematics* by Jeremy Kepner [Kep17]. The GraphBLAS C API Specification is available at http://graphblas.org. This version of SuiteSparse:GraphBLAS conforms to Version 2.0.0 (Nov 15, 2021) of *The GraphBLAS C API specification*.

In this User Guide, aspects of the GraphBLAS specification that would be true for any GraphBLAS implementation are simply called "GraphBLAS." Details unique to this particular implementation are referred to as SuiteSparse:GraphBLAS.

All functions, objects, and macros with a name of the form `GxB_*` are SuiteSparse-specific extensions to the specification.

SPEC: Non-obvious deviations or additions to the GraphBLAS C API Specification are highlighted in a box like this one, except for `GxB*` methods. They are not highlighted since their name makes it clear that they are extensions to the GraphBLAS C API.

1.1 Release Notes

- Version 6.2.0 (Feb 14, 2022)

  - added the `GxB_Iterator` object and its methods. See Section 12.
  - `@GrB` interface: revised sparse-times-full rule for the conventional semiring (the syntax `C=A*B`), so that sparse-times-full results in `C` as full, but hypersparse-times-sparse is not full (typically sparse or hypersparse).

- Version 6.1.4 (Jan 12, 2022)

  - added Section 14 to User Guide: how to get the best performance out of algorithms based on GraphBLAS.
  - `cpu_features`: no longer built as a separate library, but built directly into `libgraphblas.so` and `libgraphblas.a`. Added compile-time flags to optionally disable the use of `cpu_features` completely.
  - Octave 7: port to Apple Silicon (thanks to Gábor Szárnyas).
  - min/max monoids: real case (FP32 and FP64) no longer terminal
  - `@GrB` interface: overloaded `C=A*B` syntax where one matrix is full always results in a full matrix `C`.
  - Faster `C=A*B` for sparse-times-full and full-times-sparse for `@GrB` interface.

- Version 6.1.3 (Jan 1, 2022)

  - performance: task creation for `GrB_mxm` had a minor flaw (results were correct but parallelism suffered). Performance improvement of up to 10x when nnz(A)¡¡nnz(B).

- Version 6.1.2 (Dec 31, 2021)

  - performance: revised `swap_rule` in `GrB_mxm`, which decides whether to compute `C=A*B` or `C=(B'*A')'`, and variants, resulting in up to 3x performance gain over v6.1.1 for `GrB_mxm` (observed; could be higher in other cases).

- Version 6.1.1 (Dec 28, 2021)

  - minor revision to AVX2 and AVX512f selection
  - `cpu_features/Makefile`: remove test of `list_cpu_features` so that the package can be built when cross-compiling

- Versions 6.1.0 (Dec 26, 2021)

  - added `GxB_get` options: compiler name and version.
  - added package: https://github.com/google/cpu_features, Nov 30, 2021 version.
  - performance: faster `C+=A*B` when `C` is full, `A` is bitmap/full, and `B` is sparse/hyper. Faster `C+=A'*B` when `A` is sparse/hyper, and `B` is bitmap/full.
  - (40) bug fix: deserialization of iso and empty matrices/vectors was incorrect

- Versions 6.0.2 and 5.2.2 (Nov 30, 2021)

  - (39) bug fix: `GrB_Matrix_export`: numerical values not properly exported

- Versions 6.0.1 and 5.2.1 (Nov 27, 2021)

  - v6.0.x and v5.2.x (for the same x): differ only in `GrB_wait`, `GrB_Info`, `GrB_SCMP`, and `GxB_init`.
  - (38) bug fix: `C+=A'*B` when the accum operator is the same as the monoid and C is iso-full, and `A` or `B` are hypersparse. (dot4 method).
  - performance: `GrB_select` with user-defined `GrB_IndexUnaryOp` about 2x faster.
  - performance: faster `(MIN,MAX)_(FIRSTJ,SECONDI)` semirings

- Version 6.0.0 (Nov 15, 2021)

  - this release contains only a few changes that cause a break with backward compatibility. It is otherwise identical to v5.2.0.
  - v6.0.0 is fully compliant with the v2.0 C API Specification. Three changes from the v2.0 C API Spec are not backward compatible (`GrB_*wait`, `GrB_Info`, `GrB_SCMP`). `GxB_init` has also changed.

    * `GrB_wait (object, mode)`: was `GrB_wait (&object)`.
    * `GrB_Info`: changed enum values
    * `GrB_SCMP`: removed
    * `GxB_init (mode, malloc, calloc, realloc, free, is_thread_safe)`: the last parameter, `is_thread_safe`, is deleted. The malloc, calloc, realloc, and free functions must be thread-safe.

- Version 5.2.0 (Nov 15, 2021)

  - Added for the v2.0 C API Specification: only features that are backward compatible with SuiteSparse:GraphBLAS v5.x have been added to v5.2.0:
    * `GrB_Scalar`: replaces `GxB_Scalar`, `GxB_Scalar_*` functions renamed GrB
    * `GrB_IndexUnaryOp`: new, free, fprint, wait
    * `GrB_select`: selection via `GrB_IndexUnaryOp`
    * `GrB_apply`: with `GrB_IndexUnaryOp`
    * `GrB_reduce`: reduce matrix or vector to `GrB_Scalar`
    * `GrB_assign`, `GrB_subassion`: with `GrB_Scalar` input
    * `GrB_*_extractElement_Scalar`: get `GrB_Scalar` from a matrix or vector
    * `GrB*build`: when `dup` is `NULL`, duplicates result in an error.
    * `GrB import/export`: import/export from/to user-provided arrays
    * `GrB_EMPTY_OBJECT`, `GrB_NOT_IMPLEMENTED`: error codes added
    * `GrB_*_setElement_Scalar`: set an entry in a matrix or vector, from a `GrB_Scalar`
    * `GrB_Matrix_diag`: same as `GxB_Matrix_diag (C,v,k,NULL)`
    * `GrB_*_serialize/deserialize`: with compression
    * `GrB_ONEB_T`: binary operator, $f(x, y) = 1$, the same as `GxB_PAIR_T`.
  - `GxB*import*` and `GxB*export*`: now historical; use `GxB*pack/unpack*`
  - `GxB_select`: is now historical; use `GrB_select` instead.
  - `GxB_IGNORE_DUP`: special operator for build methods only; if dup is this operator, then duplicates are ignored (not an error)
  - `GxB_IndexUnaryOp_new`: create a named index-unary operator
  - `GxB_BinaryOp_new`: create a named binary operator

  - `GxB_UnaryOp_new`: create a named unary operator
  - `GxB_Type_new`: to create a named type
  - `GxB_Type_name`: to query the name of a type
  - added `GxB_*type_name` methods to query the name of a type as a string.
  - `GxB` methods that query an object return a `GrB_type` such as `GxB_Matrix_type` are declared historical; will be kept but not recommended (use `GxB_*type_name` methods).
  - `GxB_Matrix_serialize/deserialize`: with compression; optional descriptor.
  - `GxB_Matrix_sort`, `GxB_Vector_sort`: sort a matrix or vector
  - `GxB_eWiseUnion`: like `GrB_eWiseAdd` except for how entries in $\mathbf{A} \setminus \mathbf{B}$ and $\mathbf{B} \setminus \mathbf{A}$ are computed.
  - added LZ4/LZ4HC: compression library, http://www.lz4.org (BSD 2), v1.9.3, Copyright (c) 2011-2016, Yann Collet.
  - MIS and pagerank demos: removed; MIS added to LAGraph/experimental
  - disabled free memory pool if OpenMP not available
  - (37) bug fix: ewise `C=A+B` when all matrices are full, `GBCOMPACT` not used, but `GB_control.h` disabled the operator or type. Caught by Roi Lipman, Redis.
  - (36) bug fix: `C<M>=Z` not returning `C` as iso if `Z` iso and `C` initially empty. Caught by Erik Welch, Anaconda.
  - performance improvements: `C=A*B`: sparse/hyper times bitmap/full, and visa versa, including `C += A*B` when `C` is full.

- Version 5.1.10 (Oct 27, 2021)

  - (35) bug fix: `GB_selector`; `A->plen` and `C->plen` not updated correctly. Caught by Jeffry Lovitz, Redis.

- Version 5.1.9 (Oct 26, 2021)

  - (34) bug fix: in-place test incorrect for `C+=A'*B` using dot4
  - (33) bug fix: disable free pool if OpenMP not available

- Version 5.1.8 (Oct 5, 2021)

  - (32) bug fix: C=A*B when A is sparse and B is iso and bitmap. Caught by Mark Blanco, CMU.

- Version 5.1.7 (Aug 23, 2021)

  - (31) bug fix: `GrB_apply`, when done in-place and matrix starts non-iso and becomes iso, gave the wrong iso result. Caught by Fabian Murariu.

- Version 5.1.6 (Aug 16, 2021)

  - one-line change to `C=A*B`: faster symbolic analysis when a vector `C(:,j)` is dense (for CSC) or `C(i,:)` for CSR.

- Version 5.1.5 (July 15, 2021)

  - submission to ACM Transactions on Mathematical Software as a Collected Algorithm of the ACM.

- Version 5.1.4 (July 6, 2021)

  - faster Octave interface. Octave v7 or later is required.
  - (30) bug fix: 1-based printing not enabled for pending tuples. Caught by Will Kimmerer, while working on the Julia interface.

- Version 5.1.3 (July 3, 2021)

  - added `GxB_Matrix_iso` and `GxB_Vector_iso`: to query if a matrix or vector is held as iso-valued
  - (29) bug fix: `Matrix_pack_*R` into a matrix previously held by column, or `Matrix_pack*C` into a matrix by row, would flip the dimensions. Caught by Erik Welch, Anaconda.
  - (28) bug fix: `kron(A,B)` with iso input matrices `A` and `B` fixed. Caught by Michel Pelletier, Graphegon.
  - (27) bug fix: v5.1.0 had a wrong version of a file; posted by mistake. Caught by Michel Pelletier, Graphegon.

- Version 5.1.2 (June 30, 2021)

  - iso matrices added: these are matrices and vectors whose values in the sparsity pattern are all the same. This is an internal change to the opaque data structures of the `GrB_Matrix` and `GrB_Vector` with very little change to the API.
  - added `GxB_Matrix_build_Scalar` and `GxB_Vector_build_Scalar`, which always build iso matrices and vectors.
  - import/export methods can now import/export iso matrices and vectors.
  - added `GrB.argmin/argmax` to Octave/MATLAB interface
  - added `GxB_*_pack/unpack` methods as alternatives to import/export.
  - added `GxB_PRINT_1BASED` to the global settings.
  - added `GxB_*_memoryUsage`
  - port to Octave: `gbmake` and `gbtest` work in Octave7 to build and test the `@GrB` interface to GraphBLAS. Octave 7.0.0 is required.

- Version 5.0.6 (May 24, 2021)

  - BFS and triangle counting demos removed from GraphBLAS/Demo: see LAGraph for these algorithms. Eventually, all of GraphBLAS/Demo will be deleted, once LAGraph includes all the methods included there.

- Version 5.0.5 (May 17, 2021)

  - (26) performance bug fix: reduce-to-vector where `A` is hypersparse CSR with a transposed descriptor (or CSC with no transpose), and some cases for `GrB_mxm/mxv/vxm` when computing `C=A*B` with A hypersparse CSC and `B` bitmap/full (or `A` bitmap/full and `B` hypersparse CSR), the wrong internal method was being selected via the auto-selection strategy, resulting in a significant slowdown in some cases.

- Version 5.0.4 (May 13, 2021)

  - `@GrB` Octave/MATLAB interface: changed license to GNU General Public License v3.0 or later.

- Version 5.0.3 (May 12, 2021)

  - (25) bug fix: disabling `ANY_PAIR` semirings by editing `Source/GB_control.h` would cause a segfault if those disabled semirings were used.
  - demos are no longer built by default
  - (24) bug fix: new functions in v5.0.2 not declared as `extern` in `GraphBLAS.h`.
  - `GrB_Matrix_reduce_BinaryOp` reinstated from v4.0.3; same limit on built-in ops that correspond to known monoids.

- Version 5.0.2 (May 5, 2021)

  - (23) bug fix: `GrB_Matrix_apply_BinaryOp1st` and `2nd` were using the wrong descriptors for `GrB_INP0` and `GrB_INP1`. Caught by Erik Welch, Anaconda.
  - memory pool added for faster allocation/free of small blocks
  - `@GrB` interface ported to MATLAB R2021a.
  - `GxB_PRINTF` and `GxB_FLUSH` global options added.
  - `GxB_Matrix_diag`: construct a diagonal matrix from a vector
  - `GxB_Vector_diag`: extract a diagonal from a matrix
  - `concat/split`: methods to concatenate and split matrices.
  - `import/export`: size of arrays now in bytes, not entries. This change is required for better internal memory management, and it is not backward compatible with the `GxB*import/export` functions in v4.0. A new parameter, `is_uniform`, has been added to all import/export methods, which indicates that the matrix values are all the same.
  - (22) bug fix: SIMD vectorization was missing `reduction(+,task_cnvals)` in `GB_dense_subassign_06d_template.c`. Caught by Jeff Huang, Texas A&M, with his software package for race-condition detection.
  - `GrB_Matrix_reduce_BinaryOp`: removed. Use a monoid instead, with `GrB_reduce` or `GrB_Matrix_reduce_Monoid`.

- Version 4.0.3 (Jan 19, 2021)

  - faster min/max monoids
  - `G=GrB(G)` converts `G` from v3 object to v4

- Version 4.0.2 (Jan 13, 2021)

  - ability to load `*.mat` files saved with the v3 `GrB`

- Version 4.0.1 (Jan 4, 2021)

  - significant performance improvements: compared with v3.3.3, up to 5x faster in breadth-first-search (using `LAGraph_bfs_parent2`), and 2x faster in Betweenness-Centrality (using `LAGraph_bc_batch5`).
  - `GrB_wait(void)`, with no inputs: removed
  - `GrB_wait(&object)`: polymorphic function added
  - `GrB_*_nvals`: no longer guarantees completion; use `GrB_wait(&object)` or non-polymorphic `GrB_*_wait (&object)` instead
  - `GrB_error`: now has two parameters: a string (`char **`) and an object.
  - `GrB_Matrix_reduce_BinaryOp` limited to built-in operators that correspond to known monoids.
  - `GrB_*_extractTuples`: may return indices out of order
  - removed internal features: GBI iterator, slice and hyperslice matrices
  - bitmap/full matrices and vectors added
  - positional operators and semirings: `GxB_FIRSTI_INT32` and related ops
  - jumbled matrices: sort left pending, like zombies and pending tuples
  - `GxB_get/set`: added `GxB_SPARSITY_*` (hyper, sparse, bitmap, or full) and `GxB_BITMAP_SWITCH`.
  - `GxB_HYPER`: enum renamed to `GxB_HYPER_SWITCH`
  - `GxB*import/export`: API modified
  - `GxB_SelectOp`: `nrows` and `ncols` removed from function signature.
  - OpenMP tasking removed from mergesort and replaced with parallel for loops. Just as fast on Linux/Mac; now the performance ports to Windows.
  - `GxB_BURBLE` added as a supported feature. This was an undocumented feature of prior versions.
  - bug fix: `A({lo,hi})=scalar` `A(lo:hi)=scalar` was OK

- Version 3.3.3 (July 14, 2020). Bug fix: `w<m>=A*u` with mask non-empty

and u empty.

- Version 3.3.2 (July 3, 2020). Minor changes to build system.

- Version 3.3.1 (June 30, 2020). Bug fix to `GrB_assign` and `GxB_subassign` when the assignment is simple (`C=A`) but with typecasting.

- Version 3.3.0 (June 26, 2020). Compliant with V1.3 of the C API (except that the polymorphic `GrB_wait(&object)` doesn't appear yet; it will appear in V4.0).

  Added complex types (`GxB_FC32` and `GxB_FC64`), many unary operators, binary operators, monoids, and semirings. Added bitwise operators, and their monoids and semirings. Added the predefined monoids and semirings from the v1.3 specification. `@GrB` interface: added complex matrices and operators, and changed behavior of integer operations to more closely match the behavior on built-in integer matrices. The rules for typecasting large floating point values to integers has changed. The specific object-based `GrB_Matrix_wait`, `GrB_Vector_wait`, etc, functions have been added. The no-argument `GrB_wait()` is deprecated. Added `GrB_getVersion`, `GrB_Matrix_resize`, `GrB_Vector_resize`, `GrB_kronecker`, `GrB_*_wait`, scalar binding with binary operators for `GrB_apply`,
  `GrB_Matrix_removeElement`, and `GrB_Vector_removeElement`.

- Version 3.2.0 (Feb 20, 2020). Faster `GrB_mxm`, `GrB_mxv`, and `GrB_vxm`, and faster operations on dense matrices/vectors. Removed compile-time user objects (`GxB_*_define`), since these were not compatible with the faster matrix operations. Added the `ANY` and `PAIR` operators. Added the predefined descriptors, `GrB_DESC_*`. Added the structural mask option. Changed default chunk size to 65,536. `@GrB` interface modified: `GrB.init` is now optional.

- Version 3.1.2 (Dec, 2019). Changes to allow SuiteSparse:GraphBLAS to be compiled with the Microsoft Visual Studio compiler. This compiler does not support the `_Generic` keyword, so the polymorphic functions are not available. Use the equivalent non-polymorphic functions instead, when compiling GraphBLAS with MS Visual Studio. In addition, variable-length arrays are not supported, so user-defined types

are limited to 128 bytes in size. These changes have no effect if you have an ANSI C11 compliant compiler.

`@GrB` interface modified: `GrB.init` is now required.

- Version 3.1.0 (Oct 1, 2019). `@GrB` interface added. See the `GraphBLAS/GraphBLAS` folder for details and documentation, and Section 3.1.

- Version 3.0 (July 26, 2019), with OpenMP parallelism.

  The version number is increased to 3.0, since this version is not backward compatible with V2.x. The `GxB_select` operation changes; the `Thunk` parameter was formerly a `const void *` pointer, and is now a `GxB_Scalar`. A new parameter is added to `GxB_SelectOp_new`, to define the expected type of `Thunk`. A new parameter is added to `GxB_init`, to specify whether or not the user-provided memory management functions are thread safe.

  The remaining changes add new features, and are upward compatible with V2.x. The major change is the addition of OpenMP parallelism. This addition has no effect on the API, except that round-off errors can differ with the number of threads used, for floating-point types. `GxB_set` can optionally define the number of threads to use (the default is `omp_get_max_threads`). The number of threads can also defined globally, and/or in the `GrB_Descriptor`. The `RDIV` and `RMINUS` operators are added, which are defined as $f(x, y) = y/x$ and $f(x, y) = y - x$, respectively. Additional options are added to `GxB_get`.

- Version 2.3.3 (May 2019): Collected Algorithm of the ACM. No changes from V2.3.2 other than the documentation.

- Version 2.3 (Feb 2019) improves the performance of many GraphBLAS operations, including an early-exit for monoids. These changes have a significant impact on breadth-first-search (a performance bug was also fixed in the two BFS `Demo` codes). The matrix and vector import/export functions were added (Section 6.11), in support of the new LAGraph project (https://github.com/GraphBLAS/LAGraph, see also Section 15.1). LAGraph includes a push-pull BFS in GraphBLAS that is faster than two versions in the `Demo` folder. `GxB_init` was added to allow the memory manager functions (`malloc`, etc) to be specified.

- Version 2.2 (Nov 2018) adds user-defined objects at compile-time, via user `*.m4` files placed in `GraphBLAS/User`, which use the `GxB_*_define` macros (NOTE: feature removed in v3.2). The default matrix format is now `GxB_BY_ROW`. Also added are the `GxB_*print` methods for printing the contents of each GraphBLAS object (Section 11). PageRank demos have been added to the `Demos` folder.

- Version 2.1 (Oct 2018) was a major update with support for new matrix formats (by row or column, and hypersparse matrices), and colon notation (`I=begin:end` or `I=begin:inc:end`). Some graph algorithms are more naturally expressed with matrices stored by row, and this version includes the new `GxB_BY_ROW` format. The default format in Version 2.1 and prior versions is by column. New extensions to GraphBLAS in this version include `GxB_get`, `GxB_set`, and `GxB_AxB_METHOD`, `GxB_RANGE`, `GxB_STRIDE`, and `GxB_BACKWARDS`, and their related definitions, described in Sections 6.14, 8, and 9.

- Version 2.0 (March 2018) addressed changes in the GraphBLAS C API Specification and added `GxB_kron` and `GxB_resize`.

- Version 1.1 (Dec 2017) primarily improved the performance.

- Version 1.0 was released on Nov 25, 2017.

1.1.1 Regarding historical and deprecated functions and symbols

When a `GxB*` function or symbol is added to the C API Specification with a `GrB*` name, the new `GrB*` name should be used instead, if possible. However, the old `GxB*` name will be kept as long as possible for historical reasons. Historical functions and symbols will not always be documented here in the SuiteSparse:GraphBLAS User Guide, but they will be kept in `GraphbBLAS.h` and kept in good working order in the library. Historical functions and symbols would only be removed in the very unlikely case that they cause a serious conflict with future methods.

The only methods that have been fully deprecated and removed are the older versions of `GrB_wait` and `GrB_error` methods, which are incompatible with the latest versions.

2 Basic Concepts

Since the *GraphBLAS C API Specification* provides a precise definition of GraphBLAS, not every detail of every function is provided here. For example, some error codes returned by GraphBLAS are self-explanatory, but since a specification must precisely define all possible error codes a function can return, these are listed in detail in the *GraphBLAS C API Specification*. However, including them here is not essential and the additional information on the page might detract from a clearer view of the essential features of the GraphBLAS functions.

This User Guide also assumes the reader is familiar with Octave/MATLAB. MATLAB supports only the conventional plus-times semiring on sparse double and complex matrices, but a MATLAB-like notation easily extends to the arbitrary semirings used in GraphBLAS. The matrix multiplication in the example in the Introduction can be written in MATLAB notation as `C=A*B`, if the Boolean `OR-AND` semiring is understood. Relying on a MATLAB-like notation allows the description in this User Guide to be expressive, easy to understand, and terse at the same time. *The GraphBLAS C API Specification* also makes use of some MATLAB-like language, such as the colon notation.

MATLAB notation will always appear here in fixed-width font, such as `C=A*B(:,j)`. In standard mathematical notation it would be written as the matrix-vector multiplication $\mathbf{C} = \mathbf{A}\mathbf{b}_j$ where $\mathbf{b}_j$ is the jth column of the matrix $\mathbf{B}$. The GraphBLAS standard is a C API and SuiteSparse:GraphBLAS is written in C, and so a great deal of C syntax appears here as well, also in fixed-width font. This User Guide alternates between all three styles as needed.

2.1 Graphs and sparse matrices

Graphs can be huge, with many nodes and edges. A dense adjacency matrix $\mathbf{A}$ for a graph of n nodes takes $O(n^2)$ memory, which is impossible if n is, say, a million. Let $|\mathbf{A}|$ denote the number of entries in a matrix. Most graphs arising in practice are sparse, however, with only $|\mathbf{A}| = O(n)$ edges, where $|\mathbf{A}|$ denotes the number of edges in the graph, or the number of explicit entries present in the data structure for the matrix $\mathbf{A}$. Sparse graphs with millions of nodes and edges can easily be created by representing them as sparse matrices, where only explicit values need to be stored. Some graphs

are *hypersparse*, with $|\mathbf{A}| << n$. SuiteSparse:GraphBLAS supports three kinds of sparse matrix formats: a regular sparse format, taking $O(n + |\mathbf{A}|)$ space, a hypersparse format taking only $O(|\mathbf{A}|)$ space, and a bitmap form, taking $O(n^2)$ space. Full matrices are also represented in $O(n^2)$ space. Using its hypersparse format, creating a sparse matrix of size n-by-n where $n = 2^{60}$ (about 10^{18}) can be done on quite easily on a commodity laptop, limited only by $|\mathbf{A}|$. To the GraphBLAS user application, all matrices look alike, since these formats are opaque, and SuiteSparse:GraphBLAS switches between them at will.

A sparse matrix data structure only stores a subset of the possible n^2 entries, and it assumes the values of entries not stored have some implicit value. In conventional linear algebra, this implicit value is zero, but it differs with different semirings. Explicit values are called *entries* and they appear in the data structure. The *pattern* (also called the *structure*) of a matrix defines where its explicit entries appear. It will be referenced in one of two equivalent ways. It can be viewed as a set of indices (i, j), where (i, j) is in the pattern of a matrix $\mathbf{A}$ if $\mathbf{A}(i, j)$ is an explicit value. It can also be viewed as a Boolean matrix $\mathbf{S}$ where $\mathbf{S}(i, j)$ is true if (i, j) is an explicit entry and false otherwise. In MATLAB notation, `S=spones(A)` or `S=(A~=0)`, if the implicit value is zero. The `(i,j)` pairs, and their values, can also be extracted from the matrix via the MATLAB expression `[I,J,X]=find(A)`, where the `k`th tuple `(I(k),J(k),X(k))` represents the explicit entry `A(I(k),J(k))`, with numerical value `X(k)` equal to a_{ij}, with row index i=`I(k)` and column index j=`J(k)`.

The entries in the pattern of $\mathbf{A}$ can take on any value, including the implicit value, whatever it happens to be. This differs slightly from MATLAB, which always drops all explicit zeros from its sparse matrices. This is a minor difference but GraphBLAS cannot drop explicit zeros. For example, in the max-plus tropical algebra, the implicit value is negative infinity, and zero has a different meaning. Here, the MATLAB notation used will assume that no explicit entries are ever dropped because their explicit value happens to match the implicit value.

Graph Algorithms in the Language on Linear Algebra, Kepner and Gilbert, eds., provides a framework for understanding how graph algorithms can be expressed as matrix computations [KG11]. For additional background on sparse matrix algorithms, see also [Dav06] and [DRSL16].

2.2 Overview of GraphBLAS methods and operations

GraphBLAS provides a collection of *methods* to create, query, and free its of objects: sparse matrices, sparse vectors, scalars, types, operators, monoids, semirings, and a descriptor object used for parameter settings. Details are given in Section 6. Once these objects are created they can be used in mathematical *operations* (not to be confused with the how the term *operator* is used in GraphBLAS). A short summary of these operations and their nearest Octave/MATLAB analog is given in the table below.

operation	approximate Octave/MATLAB analog
matrix multiplication	`C=A*B`
element-wise operations	`C=A+B` and `C=A.*B`
reduction to a vector or scalar	`s=sum(A)`
apply unary operator	`C=-A`
transpose	`C=A'`
submatrix extraction	`C=A(I,J)`
submatrix assignment	`C(I,J)=A`
select	`C=tril(A)`

GraphBLAS can do far more than what Octave/MATLAB can do in these rough analogs, but the list provides a first step in describing what GraphBLAS can do. Details of each GraphBLAS operation are given in Section 10. With this brief overview, the full scope of GraphBLAS extensions of these operations can now be described.

SuiteSparse:GraphBLAS has 13 built-in scalar types: Boolean, single and double precision floating-point (real and complex), and 8, 16, 32, and 64-bit signed and unsigned integers. In addition, user-defined scalar types can be created from nearly any C `typedef`, as long as the entire type fits in a fixed-size contiguous block of memory (of arbitrary size). All of these types can be used to create GraphBLAS sparse matrices, vectors, or scalars.

The scalar addition of conventional matrix multiplication is replaced with a *monoid*. A monoid is an associative and commutative binary operator `z=f(x,y)` where all three domains are the same (the types of `x`, `y`, and `z`), and where the operator has an identity value `id` such that `f(x,id)=f(id,x)=x`. Performing matrix multiplication with a semiring uses a monoid in place of the "add" operator, scalar addition being just one of many possible monoids. The identity value of addition is zero, since $x + 0 = 0 + x = x$. GraphBLAS includes many built-in operators suitable for use as a monoid: min

(with an identity value of positive infinity), max (whose identity is negative infinity), add (identity is zero), multiply (with an identity of one), four logical operators: AND, OR, exclusive-OR, and Boolean equality (XNOR), four bitwise operators (AND, OR, XOR, and XNOR), and the ANY operator See Section 6.3.8 for more details on the unusual ANY operator. User-created monoids can be defined with any associative and commutative operator that has an identity value.

Finally, a semiring can use any built-in or user-defined binary operator `z=f(x,y)` as its "multiply" operator, as long as the type of its output, `z` matches the type of the semiring's monoid. The user application can create any semiring based on any types, monoids, and multiply operators, as long these few rules are followed.

Just considering built-in types and operators, GraphBLAS can perform `C=A*B` in thousands of unique semirings. With typecasting, any of these semirings can be applied to matrices `C`, `A`, and `B` of 13 predefined types, in any combination. This results in millions of possible kinds of sparse matrix multiplication supported by GraphBLAS, and this is counting just built-in types and operators. By contrast, MATLAB provides just two semirings for its sparse matrix multiplication `C=A*B`: plus-times-double and plus-times-complex, not counting the typecasting that MATLAB does when multiplying a real matrix times a complex matrix.

A monoid can also be used in a reduction operation, like `s=sum(A)` in MATLAB. MATLAB provides the plus, times, min, and max reductions of a real or complex sparse matrix as `s=sum(A)`, `s=prod(A)`, `s=min(A)`, and `s=max(A)`, respectively. In GraphBLAS, any monoid can be used (min, max, plus, times, AND, OR, exclusive-OR, equality, bitwise operators, or any user-defined monoid on any user-defined type).

Element-wise operations are also expanded from what can be done in MATLAB. Consider matrix addition, `C=A+B` in MATLAB. The pattern of the result is the set union of the pattern of `A` and `B`. In GraphBLAS, any binary operator can be used in this set-union "addition." The operator is applied to entries in the intersection. Entries in `A` but not `B`, or visa-versa, are copied directly into `C`, without any application of the binary operator. The accumulator operation for $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ described in Section 2.3 is one example of this set-union application of an arbitrary binary operator.

Consider element-wise multiplication, `C=A.*B` in MATLAB. The operator (multiply in this case) is applied to entries in the set intersection, and the pattern of `C` just this set intersection. Entries in `A` but not `B`, or visa-versa,

do not appear in `C`. In GraphBLAS, any binary operator can be used in this manner, not just scalar multiplication. The difference between element-wise "add" and "multiply" is not the operators, but whether or not the pattern of the result is the set union or the set intersection. In both cases, the operator is only applied to the set intersection.

Finally, GraphBLAS includes a *non-blocking* mode where operations can be left pending, and saved for later. This is very useful for submatrix assignment (`C(I,J)=A` where `I` and `J` are integer vectors), or scalar assignment (`C(i,j)=x` where `i` and `j` are scalar integers). Because of how MATLAB stores its matrices, adding and deleting individual entries is very costly. For example, this is very slow in MATLAB, taking $O(nz^2)$ time:

```
A = sparse (m,n) ;   % an empty sparse matrix
for k = 1:nz
    compute a value x, row index i, and column index j
    A (i,j) = x ;
end
```

The above code is very easy read and simple to write, but exceedingly slow. In MATLAB, the method below is preferred and is far faster, taking at most $O(|\mathbf{A}| \log |\mathbf{A}| + n)$ time. It can easily be a million times faster than the method above. Unfortunately the second method below is a little harder to read and a little less natural to write:

```
I = zeros (nz,1) ;
J = zeros (nz,1) ;
X = zeros (nz,1) ;
for k = 1:nz
    compute a value x, row index i, and column index j
    I (k) = i ;
    J (k) = j ;
    X (k) = x ;
end
A = sparse (I,J,X,m,n) ;
```

GraphBLAS can do both methods. SuiteSparse:GraphBLAS stores its matrices in a format that allows for pending computations, which are done later in bulk, and as a result it can do both methods above equally as fast as the MATLAB `sparse` function, allowing the user to write simpler code.

2.3 The accumulator and the mask

Most GraphBLAS operations can be modified via transposing input matrices, using an accumulator operator, applying a mask or its complement, and by clearing all entries the matrix `C` after using it in the accumulator operator but before the final results are written back into it. All of these steps are optional, and are controlled by a descriptor object that holds parameter settings (see Section 6.14) that control the following options:

- the input matrices `A` and/or `B` can be transposed first.
- an accumulator operator can be used, like the plus in the statement `C=C+A*B`. The accumulator operator can be any binary operator, and an element-wise "add" (set union) is performed using the operator.
- an optional *mask* can be used to selectively write the results to the output. The mask is a sparse Boolean matrix `Mask` whose size is the same size as the result. If `Mask(i,j)` is true, then the corresponding entry in the output can be modified by the computation. If `Mask(i,j)` is false, then the corresponding in the output is protected and cannot be modified by the computation. The `Mask` matrix acts exactly like logical matrix indexing in MATLAB, with one minor difference: in GraphBLAS notation, the mask operation is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$, where the mask $\mathbf{M}$ appears only on the left-hand side. In MATLAB, it would appear on both sides as `C(Mask)=Z(Mask)`. If no mask is provided, the `Mask` matrix is implicitly all true. This is indicated by passing the value `GrB_NULL` in place of the `Mask` argument in GraphBLAS operations.

This process can be described in mathematical notation as:

$\mathbf{A} = \mathbf{A}^{\mathsf{T}}$, if requested via descriptor (first input option)
$\mathbf{B} = \mathbf{B}^{\mathsf{T}}$, if requested via descriptor (second input option)
$\mathbf{T}$ is computed according to the specific operation
$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, accumulating and writing the results back via the mask

The application of the mask and the accumulator operator is written as $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$ where $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ denotes the application of the accumulator operator, and $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ denotes the mask operator via the Boolean matrix $\mathbf{M}$. The Accumulator Phase, $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$, is performed as follows:

Accumulator Phase: compute $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$:

if `accum` is `NULL`

$\mathbf{Z} = \mathbf{T}$

else

$\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$

The accumulator operator is $\odot$ in GraphBLAS notation, or `accum` in the code. The pattern of $\mathbf{C} \odot \mathbf{T}$ is the set union of the patterns of $\mathbf{C}$ and $\mathbf{T}$, and the operator is applied only on the set intersection of $\mathbf{C}$ and $\mathbf{T}$. Entries in neither the pattern of $\mathbf{C}$ nor $\mathbf{T}$ do not appear in the pattern of $\mathbf{Z}$. That is:

for all entries (i, j) in $\mathbf{C} \cap \mathbf{T}$ (that is, entries in both $\mathbf{C}$ and $\mathbf{T}$)

$z_{ij} = c_{ij} \odot t_{ij}$

for all entries (i, j) in $\mathbf{C} \setminus \mathbf{T}$ (that is, entries in $\mathbf{C}$ but not $\mathbf{T}$)

$z_{ij} = c_{ij}$

for all entries (i, j) in $\mathbf{T} \setminus \mathbf{C}$ (that is, entries in $\mathbf{T}$ but not $\mathbf{C}$)

$z_{ij} = t_{ij}$

The Accumulator Phase is followed by the Mask/Replace Phase, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ as controlled by the `GrB_REPLACE` and `GrB_COMP` descriptor options:

Mask/Replace Phase: compute $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$:

if (`GrB_REPLACE`) delete all entries in $\mathbf{C}$

if `Mask` is `NULL`

if (`GrB_COMP`)

$\mathbf{C}$ is not modified

else

$\mathbf{C} = \mathbf{Z}$

else

if (`GrB_COMP`)

$\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{Z}$

else

$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$

Both phases of the accum/mask process are illustrated in MATLAB notation in Figure 1.

A GraphBLAS operation starts with its primary computation, producing a result `T`; for matrix multiply, `T=A*B`, or if `A` is transposed first, `T=A'*B`, for example. Applying the accumulator, mask (or its complement) to obtain the final result matrix `C` can be expressed in the MATLAB `accum_mask` function

```
function C = accum_mask (C, Mask, accum, T, C_replace, Mask_complement)
[m n] = size (C.matrix) ;
Z.matrix  = zeros (m, n) ;
Z.pattern = false (m, n) ;

if (isempty (accum))
   Z = T ;     % no accum operator
else
   % Z = accum (C,T), like Z=C+T but with an binary operator, accum
   p =  C.pattern &  T.pattern ; Z.matrix (p) = accum (C.matrix (p), T.matrix (p));
   p =  C.pattern & ~T.pattern ; Z.matrix (p) = C.matrix (p) ;
   p = ~C.pattern &  T.pattern ; Z.matrix (p) = T.matrix (p) ;
   Z.pattern = C.pattern | T.pattern ;
end

% apply the mask to the values and pattern
C.matrix  = mask (C.matrix,  Mask, Z.matrix,  C_replace, Mask_complement) ;
C.pattern = mask (C.pattern, Mask, Z.pattern, C_replace, Mask_complement) ;
end

function C = mask (C, Mask, Z, C_replace, Mask_complement)
% replace C if requested
if (C_replace)
   C (:,:) = 0 ;
end
if (isempty (Mask))             % if empty, Mask is implicit ones(m,n)
   % implicitly, Mask = ones (size (C))
   if (~Mask_complement)
      C = Z ;                   % this is the default
   else
      C = C ;                   % Z need never have been computed
   end
else
   % apply the mask
   if (~Mask_complement)
      C (Mask) = Z (Mask) ;
   else
      C (~Mask) = Z (~Mask) ;
   end
end
end
```

Figure 1: Applying the mask and accumulator, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$

shown in the figure. This function is an exact, fully functional, and nearly-complete description of the GraphBLAS accumulator/mask operation. The only aspects it does not consider are typecasting (see Section 2.4), and the value of the implicit identity (for those, see another version in the `Test` folder).

One aspect of GraphBLAS cannot be as easily expressed in a MATLAB sparse matrix: namely, what is the implicit value of entries not in the pattern? To accommodate this difference in the `accum_mask` MATLAB function, each sparse matrix `A` is represented with its values `A.matrix` and its pattern, `A.pattern`. The latter could be expressed as the sparse matrix `A.pattern=spones(A)` or `A.pattern=(A~=0)` in MATLAB, if the implicit value is zero. With different semirings, entries not in the pattern can be `1`, `+Inf`, `-Inf`, or whatever is the identity value of the monoid. As a result, Figure 1 performs its computations on two MATLAB matrices: the values in `A.matrix` and the pattern in the logical matrix `A.pattern`. Implicit values are untouched.

The final computation in Figure 1 with a complemented `Mask` is easily expressed in MATLAB as `C(~Mask)=Z(~Mask)` but this is costly if `Mask` is very sparse (the typical case). It can be computed much faster in MATLAB without complementing the sparse `Mask` via:

```
R = Z ; R (Mask) = C (Mask) ; C = R ;
```

A set of MATLAB functions that precisely compute the $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$ operation according to the full GraphBLAS specification is provided in SuiteSparse:GraphBLAS as `GB_spec_accum.m`, which computes $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$, and `GB_spec_mask.m`, which computes $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$. SuiteSparse:GraphBLAS includes a complete list of `GB_spec_*` functions that illustrate every GraphBLAS operation.

The methods in Figure 1 rely heavily on MATLAB's logical matrix indexing. For those unfamiliar with logical indexing in MATLAB, here is short summary. Logical matrix indexing in MATLAB is written as `A(Mask)` where `A` is any matrix and `Mask` is a logical matrix the same size as `A`. The expression `x=A(Mask)` produces a column vector `x` consisting of the entries of `A` where `Mask` is true. On the left-hand side, logical submatrix assignment `A(Mask)=x` does the opposite, copying the components of the vector `x` into the places in `A` where `Mask` is true. For example, to negate all values greater than 10 using logical indexing in MATLAB:

```
>> A = magic (4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> A (A>10) = - A (A>10)
A =
   -16     2     3   -13
     5   -11    10     8
     9     7     6   -12
     4   -14   -15     1
```

In MATLAB, logical indexing with a sparse matrix `A` and sparse logical matrix `Mask` is a built-in method. The Mask operator in GraphBLAS works identically as sparse logical indexing in MATLAB, but is typically far faster in SuiteSparse:GraphBLAS than the same operation using MATLAB sparse matrices.

2.4 Typecasting

If an operator `z=f(x)` or `z=f(x,y)` is used with inputs that do not match its inputs `x` or `y`, or if its result `z` does not match the type of the matrix it is being stored into, then the values are typecasted. Typecasting in GraphBLAS extends beyond just operators. Almost all GraphBLAS methods and operations are able to typecast their results, as needed.

If one type can be typecasted into the other, they are said to be *compatible*. All built-in types are compatible with each other. GraphBLAS cannot typecast user-defined types thus any user-defined type is only compatible with itself. When GraphBLAS requires inputs of a specific type, or when one type cannot be typecast to another, the GraphBLAS function returns an error code, `GrB_DOMAIN_MISMATCH` (refer to Section 5.6 for a complete list of error codes). Typecasting can only be done between built-in types, and it follows the rules of the ANSI C language (not MATLAB) wherever the rules of ANSI C are well-defined.

However, unlike MATLAB, the ANSI C11 language specification states that the results of typecasting a `float` or `double` to an integer type is not always defined. In SuiteSparse:GraphBLAS, whenever C leaves the result undefined the rules used in MATLAB are followed. In particular `+Inf` converts to the largest integer value, `-Inf` converts to the smallest (zero for unsigned

integers), and NaN converts to zero. Positive values outside the range of the integer are converted to the largest positive integer, and negative values less than the most negative integer are converted to that most negative integer. Other than these special cases, SuiteSparse:GraphBLAS trusts the C compiler for the rest of its typecasting.

Typecasting to bool is fully defined in the C language specification, even for NaN. The result is false if the value compares equal to zero, and true otherwise. Thus NaN converts to true. This is unlike MATLAB, which does not allow a typecast of a NaN to the MATLAB logical type.

SPEC: the GraphBLAS API C Specification states that typecasting follows the rules of ANSI C. Yet C leaves some typecasting undefined. All typecasting between built-in types in SuiteSparse:GraphBLAS is precisely defined, as an extension to the specification.

SPEC: Some functions do not make use of all of their inputs; in particular the binary operators FIRST, SECOND, and ONEB, and many of the index unary operators. The Specification requires that the inputs to these operators must be compatible with (that is, can be typecasted to) the inputs to the operators, even if those inputs are not used and no typecasting would ever occur. As an extension to the specification, SuiteSparse:GraphBLAS does not perform this error check on unused inputs of built-in operators. For example, the GrB_FIRST_INT64 operator can be used in GrB_eWiseAdd(C,..,A,B,...) on a matrix B of any type, including user-defined types. For this case, the matrix A must be compatible with GrB_INT64.

2.5 Notation and list of GraphBLAS operations

As a summary of what GraphBLAS can do, the following table lists all GraphBLAS operations. Upper case letters denote a matrix, lower case letters are vectors, and **AB** denote the multiplication of two matrices over a semiring.

Each operation takes an optional GrB_Descriptor argument that modifies the operation. The input matrices $\mathbf{A}$ and $\mathbf{B}$ can be optionally transposed, the mask $\mathbf{M}$ can be complemented, and $\mathbf{C}$ can be cleared of its entries after it is used in $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ but before the $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ assignment. Vectors are never transposed via the descriptor.

Let $\mathbf{A} \oplus \mathbf{B}$ denote the element-wise operator that produces a set union pattern (like `A+B` in MATLAB). Any binary operator can be used this way in GraphBLAS, not just plus. Let $\mathbf{A} \otimes \mathbf{B}$ denote the element-wise operator that produces a set intersection pattern (like `A.*B` in MATLAB); any binary operator can be used this way, not just times.

Reduction of a matrix $\mathbf{A}$ to a vector reduces the ith row of $\mathbf{A}$ to a scalar w_i. This is like `w=sum(A')` since by default, MATLAB reduces down the columns, not across the rows.

`GrB_mxm`	matrix-matrix multiply	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$
`GrB_vxm`	vector-matrix multiply	$\mathbf{w}^\mathsf{T}\langle\mathbf{m}^\mathsf{T}\rangle = \mathbf{w}^\mathsf{T} \odot \mathbf{u}^\mathsf{T}\mathbf{A}$
`GrB_mxv`	matrix-vector multiply	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{Au}$
`GrB_eWiseMult`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \otimes \mathbf{B})$
	set intersection	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$
`GrB_eWiseAdd`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$
	set union	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`GxB_eWiseUnion`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$
	set union	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`GrB_extract`	extract submatrix	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}(\mathbf{I}, \mathbf{J})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{u}(\mathbf{i})$
`GxB_subassign`	assign submatrix	$\mathbf{C}(\mathbf{I}, \mathbf{J})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{I}, \mathbf{J}) \odot \mathbf{A}$
	(with submask for $\mathbf{C}(\mathbf{I}, \mathbf{J})$)	$\mathbf{w}(\mathbf{i})\langle\mathbf{m}\rangle = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GrB_assign`	assign submatrix	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I}, \mathbf{J}) = \mathbf{C}(\mathbf{I}, \mathbf{J}) \odot \mathbf{A}$
	(with mask for $\mathbf{C}$)	$\mathbf{w}\langle\mathbf{m}\rangle(\mathbf{i}) = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GrB_apply`	apply unary operator	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u})$
	apply binary operator	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, y)$
		$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(x, \mathbf{A})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, y)$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(x, \mathbf{u})$
	apply index-unary op	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, i, j, k)$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, i, 0, k)$
`GrB_select`	select entries	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \text{select}(\mathbf{A}, i, j, k)$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \text{select}(\mathbf{u}, i, 0, k)$
`GrB_reduce`	reduce to vector	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$
	reduce to scalar	$s = s \odot [\oplus_{ij} \mathbf{A}(i, j)]$
`GrB_transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}^\mathsf{T}$
`GrB_kronecker`	Kronecker product	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \text{kron}(\mathbf{A}, \mathbf{B})$

3 Interfaces to Octave, MATLAB, Python, Julia, Java

The Octave/MATLAB interface to SuiteSparse:GraphBLAS is included with this distribution, described in Section 3.1. It is fully polished, and fully tested, but does have some limitations that will be addressed in future releases. Two Python interfaces are now available, as is a Julia interface. These are not part of the SuiteSparse:GraphBLAS distribution. See the links below (see Sections 3.2 and 3.3).

3.1 Octave/MATLAB Interface

An easy-to-use Octave/MATLAB interface for SuiteSparse:GraphBLAS is available; see the documentation in the `GraphBLAS/GraphBLAS` folder for details. Start with the `README.md` file in that directory. An easy-to-read output of the MATLAB demos can be found in `GraphBLAS/GraphBLAS/demo/html`.

The Octave/MATLAB interface adds the `@GrB` class, which is an opaque Octave/MATLAB object that contains a GraphBLAS matrix, either double or single precision (real or complex), boolean, or any of the built-in integer types. Octave/MATLAB sparse and full matrices can be arbitrarily mixed with GraphBLAS matrices. The following overloaded operators and methods all work as you would expect for any matrix. The matrix multiplication `A*B` uses the conventional `PLUS_TIMES` semiring.

```
A+B     A-B     A*B     A.*B    A./B    A.\B    A.^b     A/b     C=A(I,J)
-A      +A      ~A      A'      A.'     A&B     A|B      b\A     C(I,J)=A
A~=B    A>B     A==B    A<=B    A>=B    A<B     [A,B]    [A;B]   A(1:end,1:end)
```

For a list of overloaded operations and static methods, type `methods GrB` in Octave/MATLAB, or `help GrB` for more details.

Limitations: Some features for Octave/MATLAB sparse matrices are not yet available for GraphBLAS matrices. Some of these may be added in future releases.

- `GrB` matrices with dimension larger than `2^53` do not display properly in the `whos` command. The size is displayed correctly with `disp` or `display`.
- Non-blocking mode is not exploited.

- Linear indexing: `A(:)` for a 2D matrix, and `I=find(A)`.
- Singleton expansion.
- Dynamically growing arrays, where `C(i)=x` can increase the size of `C`.
- Saturating element-wise binary and unary operators for integers. For `C=A+B` with MATLAB `uint8` matrices, results saturate if they exceed 255. This is not compatible with a monoid for `C=A*B`, and thus MATLAB does not support matrix-matrix multiplication with `uint8` matrices. In GraphBLAS, `uint8` addition acts in a modulo fashion.
- Solvers, so that `x=A\b` could return a GF(2) solution, for example.
- Sparse matrices with dimension higher than 2.

3.2 Python Interface

See Michel Pelletier's Python interface at https://github.com/michelp/pygraphblas; it also appears at https://anaconda.org/conda-forge/pygraphblas.

See Jim Kitchen and Erik Welch's (both from Anaconda, Inc.) Python interface at https://github.com/metagraph-dev/grblas. See also https://anaconda.org/conda-forge/graphblas.

Both of them allow for pending work to be left pending in a `GrB_Matrix`.

3.3 Julia Interface

The Julia interface is at https://github.com/JuliaSparse/SuiteSparseGraphBLAS.jl, developed by Will Kimmerer, Abhinav Mehndiratta, Miha Zgubic, and Viral Shah. Unlike the Octave/MATLAB interface (and like the Python interfaces) the Julia interface can keep pending work (zombies, pending tuples, jumbled state) in a `GrB_Matrix`. This makes Python and Julia the best high-level interfaces for SuiteSparse:GraphBLAS. MATLAB is not as well suited, since it does not allow inputs to a function or mexFunction to be modified, so any pending work must be finished before a matrix can be used as input.

3.4 Java Interface

Fabian Murariu is working on a Java interface. See https://github.com/fabianmurariu/graphblas-java-native.

4 Performance of MATLAB versus GraphBLAS

MATLAB R2021a includes v3.3 of SuiteSparse:GraphBLAS as a built-in library, but uses it only for `C=A*B` when both `A` and `B` are sparse. In prior versions of MATLAB, `C=A*B` relied on the `SFMULT` and `SSMULT` packages in SuiteSparse, which are single-threaded (also written by this author). The GraphBLAS `GrB_mxm` is up to 30x faster on a 20-core Intel Xeon, compared with `C=A*B` in MATLAB R2020b and earlier. With MATLAB R2021a and later, the performance of `C=A*B` when using MATLAB sparse matrices is identical to the performance for GraphBLAS matrices, since the same code is being used by both (`GrB_mxm`).

Other methods in GraphBLAS are also faster, some *extremely* so, but are not yet exploited as built-in operations MATLAB. In particular, the statement `C(M)=A` (where `M` is a logical matrix) takes under a second for a large sparse problem when using GraphBLAS via its `@GrB` interface. By stark contrast, MATLAB would take about 4 or 5 days, a speedup of about 500,000x. For a smaller problem, GraphBLAS takes 0.4 seconds while MATLAB takes 28 hours (a speedup of about 250,000x). Both cases use the same statement with the same syntax (`C(M)=A`) and compute exactly the same result. Below are the results for `n`-by-`n` matrices in GraphBLAS v5.0.6 and MATLAB R2020a, on a Dell XPS13 laptop (16GB RAM, Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz with 4 hardware cores). GraphBLAS is using 4 threads.

`n`	`nnz(C)`	`nnz(M)`	GraphBLAS (sec)	MATLAB (sec)	speedup
2,048	20,432	2,048	0.005	0.024	4.7
4,096	40,908	4,096	0.003	0.115	39
8,192	81,876	8,191	0.009	0.594	68
16,384	163,789	16,384	0.009	2.53	273
32,768	327,633	32,767	0.014	12.4	864
65,536	655,309	65,536	0.025	65.9	2,617
131,072	1,310,677	131,070	0.055	276.2	4,986
262,144	2,621,396	262,142	0.071	1,077	15,172
524,288	5,242,830	524,288	0.114	5,855	51,274
1,048,576	10,485,713	1,048,576	0.197	27,196	137,776
2,097,152	20,971,475	2,097,152	0.406	100,799	248,200
4,194,304	41,942,995	4,194,304	0.855	4 to 5 days?	500,000?

The assignment `C(I,J)=A` in MATLAB, when using `@GrB` objects, is up to 1000x faster than the same statement with the same syntax, when using MATLAB sparse matrices instead. Matrix concatenation `C = [A B]` is about 17 times faster in GraphBLAS, on a 20-core Intel Xeon. For more details, see the `GraphBLAS/GraphBLAS/demo` folder and its contents.

Below is a comparison of other methods in SuiteSparse:GraphBLAS, compared with MATLAB 2021a. SuiteSparse:GraphBLAS: v6.1.4 (Jan 12, 2022), was used, compiled with gcc 11.2.0. The system is an Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz (20 hardware cores, 40 threads), Ubuntu 20.04, 256GB RAM. Full details appear in the `GraphBLAS/GraphBLAS/demo/benchmark` folder. For this matrix, SuiteSparse:GraphBLAS is anywhere from 3x to 17x faster than the built-in methods in MATLAB. This matrix is not special, but is typical of the relative performance of many large matrices. Note that two of these (`C=L*S` and `C=S*R`) rely on an older version of SuiteSparse:GraphBLAS (v3.3.3) built into MATLAB R2021a.

```
Legend:
S: large input sparse matrix (n-by-n), the GAP-twitter matrix
x: dense vector (1-by-n or n-by-1)
F: dense matrix (4-by-n or n-by-4)
L: 8-by-n sparse matrix, about 1000 entries
R: n-by-8 sparse matrix, about 1000 entries
B: n-by-n sparse matrix, about nnz(S)/10 entries
p,q: random permutation vectors

GAP/GAP-twitter: n: 61.5784 million nnz: 1468.36 million
(run time in seconds):
y=S*x    MATLAB:  22.8012 GrB:   2.4018 speedup:     9.49
y=x*S    MATLAB:  16.1618 GrB:   1.1610 speedup:    13.92
C=S*F    MATLAB:  30.6121 GrB:   9.7052 speedup:     3.15
C=F*S    MATLAB:  26.4044 GrB:   1.5245 speedup:    17.32
C=L*S    MATLAB:  19.1228 GrB:   2.4301 speedup:     7.87
C=S*R    MATLAB:   0.0087 GrB:   0.0020 speedup:     4.40
C=S'     MATLAB: 224.7268 GrB:  22.6855 speedup:     9.91
C=S+S    MATLAB:  14.3368 GrB:   1.5539 speedup:     9.23
C=S+B    MATLAB:  15.5600 GrB:   1.5098 speedup:    10.31
C=S(p,q) MATLAB:  95.6219 GrB:  15.9468 speedup:     6.00
```

5 GraphBLAS Context and Sequence

A user application that directly relies on GraphBLAS must include the `GraphBLAS.h` header file:

```
#include "GraphBLAS.h"
```

The `GraphBLAS.h` file defines functions, types, and macros prefixed with `GrB_` and `GxB_` that may be used in user applications. The prefix `GrB_` denotes items that appear in the official *GraphBLAS C API Specification*. The prefix `GxB_` refers to SuiteSparse-specific extensions to the GraphBLAS API.

The `GraphBLAS.h` file includes all the definitions required to use GraphBLAS, including the following macros that can assist a user application in compiling and using GraphBLAS.

There are two version numbers associated with SuiteSparse:GraphBLAS: the version of the *GraphBLAS C API Specification* it conforms to, and the version of the implementation itself. These can be used in the following manner in a user application:

```
#if GxB_SPEC_VERSION >= GxB_VERSION (2,0,3)
... use features in GraphBLAS specification 2.0.3 ...
#else
... only use features in early specifications
#endif

#if GxB_IMPLEMENTATION >= GxB_VERSION (5,2,0)
... use features from version 5.2.0 (or later)
of a specific GraphBLAS implementation
#endif
```

SuiteSparse:GraphBLAS also defines the following strings with `#define`. Refer to the `GraphBLAS.h` file for details.

Macro	purpose
`GxB_IMPLEMENTATION_ABOUT`	this particular implementation, copyright, and URL
`GxB_IMPLEMENTATION_DATE`	the date of this implementation
`GxB_SPEC_ABOUT`	the GraphBLAS specification for this implementation
`GxB_SPEC_DATE`	the date of the GraphBLAS specification
`GxB_IMPLEMENTATION_LICENSE`	the license for this particular implementation

Finally, SuiteSparse:GraphBLAS gives itself a unique name of the form `GxB_SUITESPARSE_GRAPHBLAS` that the user application can use in `#ifdef` tests. This is helpful in case a particular implementation provides non-standard features that extend the GraphBLAS specification, such as additional predefined built-in operators, or if a GraphBLAS implementation does not yet fully implement all of the GraphBLAS specification.

For example, SuiteSparse:GraphBLAS predefines additional built-in operators not in the specification. If the user application wishes to use these in any GraphBLAS implementation, an `#ifdef` can control when they are used. Refer to the examples in the `GraphBLAS/Demo` folder.

As another example, the GraphBLAS API states that an implementation need not define the order in which `GrB_Matrix_build` assembles duplicate tuples in its `[I,J,X]` input arrays. As a result, no particular ordering should be relied upon in general. However, SuiteSparse:GraphBLAS does guarantee an ordering, and this guarantee will be kept in future versions of SuiteSparse:GraphBLAS as well. Since not all implementations will ensure a particular ordering, the following can be used to exploit the ordering returned by SuiteSparse:GraphBLAS.

```
#ifdef GxB_SUITESPARSE_GRAPHBLAS
// duplicates in I, J, X assembled in a specific order;
// results are well-defined even if op is not associative.
GrB_Matrix_build (C, I, J, X, nvals, op) ;
#else
// duplicates in I, J, X assembled in no particular order;
// results are undefined if op is not associative.
GrB_Matrix_build (C, I, J, X, nvals, op) ;
#endif
```

The remainder of this section describes GraphBLAS functions that start or finalize GraphBLAS, error handling, and the GraphBLAS integer.

GraphBLAS function/type	purpose	Section
`GrB_Index`	the GraphBLAS integer	5.1
`GrB_init`	start up GraphBLAS	5.2
`GrB_getVersion`	C API supported by the library	5.3
`GxB_init`	start up GraphBLAS with different `malloc`	5.4
`GrB_Info`	status code returned by GraphBLAS functions	5.5
`GrB_error`	get more details on the last error	5.6
`GrB_finalize`	finish GraphBLAS	5.7

5.1 GrB_Index: the GraphBLAS integer

Matrix and vector dimensions and indexing rely on a specific integer, `GrB_Index`, which is defined in `GraphBLAS.h` as

```
typedef uint64_t GrB_Index ;
```

Row and column indices of an `nrows`-by-`ncols` matrix range from zero to the `nrows-1` for the rows, and zero to `ncols-1` for the columns. Indices are zero-based, like C, and

not one-based, like Octave/MATLAB. In SuiteSparse:GraphBLAS, the largest permitted index value is `GrB_INDEX_MAX`, defined as $2^{60} - 1$. The largest permitted matrix or vector dimension is 2^{60} (that is, `GrB_INDEX_MAX+1`). The largest `GrB_Matrix` that SuiteSparse: GraphBLAS can construct is thus 2^{60}-by-2^{60}. An n-by-n matrix $\mathbf{A}$ that size can easily be constructed in practice with $O(|\mathbf{A}|)$ memory requirements, where $|\mathbf{A}|$ denotes the number of entries that explicitly appear in the pattern of $\mathbf{A}$. The time and memory required to construct a matrix that large does not depend on n, since SuiteSparse:GraphBLAS can represent $\mathbf{A}$ in hypersparse form (see Section 8.3). The largest `GrB_Vector` that can be constructed is 2^{60}-by-1.

5.2 GrB_init: initialize GraphBLAS

```
typedef enum
{
    GrB_NONBLOCKING = 0,    // methods may return with pending computations
    GrB_BLOCKING = 1        // no computations are ever left pending
}
GrB_Mode ;
```

```
GrB_Info GrB_init           // start up GraphBLAS
(
    GrB_Mode mode           // blocking or non-blocking mode
) ;
```

`GrB_init` must be called before any other GraphBLAS operation. It defines the mode that GraphBLAS will use: blocking or non-blocking. With blocking mode, all operations finish before returning to the user application. With non-blocking mode, operations can be left pending, and are computed only when needed. Non-blocking mode can be much faster than blocking mode, by many orders of magnitude in extreme cases. Blocking mode should be used only when debugging a user application. The mode cannot be changed once it is set by `GrB_init`.

GraphBLAS objects are opaque. This allows GraphBLAS to postpone operations and then do them later in a more efficient manner by rearranging them and grouping them together. In non-blocking mode, the computations required to construct an opaque GraphBLAS object might not be finished when the GraphBLAS method or operation returns to the user. However, user-provided arrays are not opaque, and GraphBLAS methods and operations that read them (such as `GrB_Matrix_build`) or write to them (such as `GrB_Matrix_extractTuples`) always finish reading them, or creating them, when the method or operation returns to the user application.

All methods and operations that extract values from a GraphBLAS object and return them into non-opaque user arrays always ensure that the user-visible arrays are fully populated when they return: `GrB_*_reduce` (to scalar), `GrB_*_nvals`, `GrB_*_extractElement`, and `GrB_*_extractTuples`. These functions do *not* guarantee that the opaque objects they depend on are finalized. To do that, use `GrB_wait` instead.

SuiteSparse:GraphBLAS is multithreaded internally, via OpenMP, and it is also safe to use in a multithreaded user application. See Section 16 for details. User threads must not operate on the same matrices at the same time, with one exception. Multiple user threads can use the same matrices or vectors as read-only inputs to GraphBLAS operations or methods, but only if they have no pending operations (use `GrB_wait` first). User threads cannot simultaneously modify a matrix or vector via any GraphBLAS operation or method.

It is safe to use the internal parallelism in SuiteSparse:GraphBLAS on matrices, vectors, and scalars that are not yet completed. The library handles this on its own. The `GrB_wait` function is only needed when a user application makes multiple calls to GraphBLAS in parallel, from multiple user threads.

With multiple user threads, exactly one user thread must call `GrB_init` before any user thread may call any `GrB_*` or `GxB_*` function. When the user application is finished, exactly one user thread must call `GrB_finalize`, after which no user thread may call any `GrB_*` or `GxB_*` function. The mode of a GraphBLAS session can be queried with `GxB_get`; see Section 8 for details.

5.3 GrB_getVersion: determine the C API Version

```
GrB_Info GrB_getVersion         // runtime access to C API version number
(
    unsigned int *version,      // returns GRB_VERSION
    unsigned int *subversion    // returns GRB_SUBVERSION
) ;
```

GraphBLAS defines two compile-time constants that define the version of the C API Specification that is implemented by the library: `GRB_VERSION` and `GRB_SUBVERSION`. If the user program was compiled with one version of the library but linked with a different one later on, the compile-time version check with `GRB_VERSION` would be stale. `GrB_getVersion` thus provides a runtime access of the version of the C API Specification supported by the library.

5.4 GxB_init: initialize with alternate malloc

```
% in SuiteSparse:GraphBLAS v5.2.0 or earlier:
GrB_Info GxB_init           // start up GraphBLAS and also define malloc
(
    GrB_Mode mode,          // blocking or non-blocking mode
    // pointers to memory management functions.
    void * (* user_malloc_function  ) (size_t),
    void * (* user_calloc_function  ) (size_t, size_t),
    void * (* user_realloc_function ) (void *, size_t),
    void   (* user_free_function    ) (void *),
    bool ignored            // unused parameter to be removed in v6.0
) ;

% in SuiteSparse:GraphBLAS v6.0.0 or later:
GrB_Info GxB_init           // start up GraphBLAS and also define malloc
(
    GrB_Mode mode,          // blocking or non-blocking mode
    // pointers to memory management functions.
    void * (* user_malloc_function  ) (size_t),
    void * (* user_calloc_function  ) (size_t, size_t),
    void * (* user_realloc_function ) (void *, size_t),
    void   (* user_free_function    ) (void *)
) ;
```

`GxB_init` is identical to `GrB_init`, except that it also redefines the memory management functions that SuiteSparse:GraphBLAS will use. Giving the user application control over this is particularly important when using the `GxB_*pack`, `GxB_*unpack`, and `GxB_*serialize` functions described in Sections 6.10 and 6.11, since they require the user application and GraphBLAS to use the same memory manager.

`user_calloc_function` and `user_realloc_function` are optional, and may be `NULL`. If `NULL`, then the `user_malloc_function` is relied on instead, for all memory allocations.

These functions can only be set once, when GraphBLAS starts. Either `GrB_init` or `GxB_init` must be called before any other GraphBLAS operation, but not both. The functions passed to `GxB_init` must be thread-safe.

The following usage is identical to `GrB_init(mode)`:

```
% in SuiteSparse:GraphBLAS v5.2.0:
GxB_init (mode, malloc, calloc, realloc, free, true) ;
% in SuiteSparse:GraphBLAS v6.0.0:
GxB_init (mode, malloc, calloc, realloc, free) ;
```

The last parameter (`ignored`) is ignored in v5.2 and is removed in v6.0.

5.5 GrB_Info: status code returned by GraphBLAS

Each GraphBLAS method and operation returns its status to the caller as its return value, an enumerated type (an `enum`) called `GrB_Info`. The first two values in the following table denote a successful status, the rest are error codes. SuiteSparse:GraphBLAS v5.x and earlier use the enum values in the v5 column, since the C API Specification did not define them. The values are now defined in the v2.0 C API Specification, and appear in SuiteSparse:GraphBLAS v6.0.0 with the values in the v6 column.

Not all GraphBLAS methods or operations can return all status codes. In the discussions of each method and operation in this User Guide, most of the obvious error code returns are not discussed. For example, if a required input is a `NULL` pointer, then `GrB_NULL_POINTER` is returned. Only error codes specific to the method or that require elaboration are discussed here. For a full list of the status codes that each GraphBLAS function can return, refer to *The GraphBLAS C API Specification* [BMM+17b, BBM+21].

Error	v5	v6	description
GrB_SUCCESS	0	0	the method or operation was successful
GrB_NO_VALUE	1	1	the method was successful, but the entry does not appear in the matrix or vector.
GxB_EXHAUSTED	-	2	the iterator is exhausted
GrB_UNINITIALIZED_OBJECT	2	-1	object has not been initialized
GrB_NULL_POINTER	4	-2	input pointer is NULL
GrB_INVALID_VALUE	5	-3	generic error code; some value is bad
GrB_INVALID_INDEX	6	-4	a row or column index is out of bounds
GrB_DOMAIN_MISMATCH	7	-5	object domains are not compatible
GrB_DIMENSION_MISMATCH	8	-6	matrix dimensions do not match
GrB_OUTPUT_NOT_EMPTY	9	-7	output matrix already has values in it
GrB_NOT_IMPLEMENTED	-8	-8	not implemented in SS:GrB
GrB_PANIC	13	-101	unrecoverable error
GrB_OUT_OF_MEMORY	10	-102	out of memory
GrB_INSUFFICIENT_SPACE	11	-103	output array not large enough
GrB_INVALID_OBJECT	3	-104	object is corrupted
GrB_INDEX_OUT_OF_BOUNDS	12	-105	a row or column index is out of bounds
GrB_EMPTY_OBJECT	-106	-106	a input scalar has no entry

5.6 GrB_error: get more details on the last error

```
GrB_Info GrB_error      // return a string describing the last error
(
    const char **error, // error string
    <type> object       // a GrB_matrix, GrB_Vector, etc.
) ;
```

Each GraphBLAS method and operation returns a `GrB_Info` error code. The `GrB_error` function returns additional information on the error for a particular object in a null-terminated string. The string returned by `GrB_error` is never a `NULL` string, but it may have length zero (with the first entry being the `'\0'` string-termination value). The string must not be freed or modified.

```
info = GrB_some_method_here (C, ...) ;
if (! (info == GrB_SUCCESS || info == GrB_NO_VALUE))
{
    char *err ;
    GrB_error (&err, C) ;
    printf ("info: %d error: %s\n", info, err) ;
}
```

If `C` has no error status, or if the error is not recorded in the string, an empty non-null string is returned. In particular, out-of-memory conditions result in an empty string from `GrB_error`.

SuiteSparse:GraphBLAS reports many helpful details via `GrB_error`. For example, if a row or column index is out of bounds, the report will state what those bounds are. If a matrix dimension is incorrect, the mismatching dimensions will be provided. `GrB_BinaryOp_new`, `GrB_UnaryOp_new`, and `GrB_IndexUnaryOp_new` record the name the function passed to them, and `GrB_Type_new` records the name of its type parameter, and these are printed if the user-defined types and operators are used incorrectly. Refer to the output of the example programs in the `Demo` and `Test` folder, which intentionally generate errors to illustrate the use of `GrB_error`.

The only functions in GraphBLAS that return an error string are functions that have a single input/output argument `C`, as a `GrB_Matrix`, `GrB_Vector`, `GrB_Scalar`, or `GrB_Descriptor`. Methods that create these objects (such as `GrB_Matrix_new`) return a `NULL` object on failure, so these methods cannot also return an error string in `C`.

Any subsequent GraphBLAS method that modifies the object `C` clears the error string.

Note that `GrB_NO_VALUE` is an not error, but an informational status. `GrB_*_extractElment(&x,A,i,j)`, which does `x=A(i,j)`, returns this value to indicate that `A(i,j)` is not present in the matrix. That method does not have an input/output object so it cannot return an error string.

5.7 GrB_finalize: finish GraphBLAS

```
GrB_Info GrB_finalize ( ) ;     // finish GraphBLAS
```

`GrB_finalize` must be called as the last GraphBLAS operation, even after all calls to `GrB_free`. All GraphBLAS objects created by the user application should be freed first, before calling `GrB_finalize` since `GrB_finalize` will not free those objects. In non-blocking mode, GraphBLAS may leave some computations as pending. These computations can be safely abandoned if the user application frees all GraphBLAS objects it has created and then calls `GrB_finalize`. When the user application is finished, exactly one user thread must call `GrB_finalize`.

6 GraphBLAS Objects and their Methods

GraphBLAS defines ten different objects to represent matrices, vectors, scalars, data types, operators (binary, unary, and index-unary), monoids, semirings, and a *descriptor* object used to specify optional parameters that modify the behavior of a GraphBLAS operation.

The GraphBLAS API makes a distinction between *methods* and *operations*. A method is a function that works on a GraphBLAS object, creating it, destroying it, or querying its contents. An operation (not to be confused with an operator) acts on matrices and/or vectors in a semiring.

`GrB_Type`	a scalar data type
`GrB_UnaryOp`	a unary operator $z = f(x)$, where z and x are scalars
`GrB_BinaryOp`	a binary operator $z = f(x, y)$, where z, x, and y are scalars
`GrB_IndexUnaryOp`	an index-unary operator
`GrB_Monoid`	an associative and commutative binary operator and its identity value
`GrB_Semiring`	a monoid that defines the "plus" and a binary operator that defines the "multiply" for an algebraic semiring
`GrB_Matrix`	a 2D sparse matrix of any type
`GrB_Vector`	a 1D sparse column vector of any type
`GrB_Scalar`	a scalar of any type
`GrB_Descriptor`	a collection of parameters that modify an operation

Each of these objects is implemented in C as an opaque handle, which is a pointer to a data structure held by GraphBLAS. User applications may not examine the content of the object directly; instead, they can pass the handle back to GraphBLAS which will do the work. Assigning one handle to another is valid but it does not make a copy of the underlying object.

6.1 The GraphBLAS type: GrB_Type

A GraphBLAS `GrB_Type` defines the type of scalar values that a matrix or vector contains, and the type of scalar operands for a unary or binary operator. There are 13 built-in types, and a user application can define any types of its own as well. The built-in types correspond to built-in types in C (in the `#include` files `stdbool.h`, `stdint.h`, and `complex.h`) as listed in the following table.

GraphBLAS type	C type	description	range
`GrB_BOOL`	`bool`	Boolean	true (1), false (0)
`GrB_INT8`	`int8_t`	8-bit signed integer	-128 to 127
`GrB_INT16`	`int16_t`	16-bit integer	-2^{15} to $2^{15}-1$
`GrB_INT32`	`int32_t`	32-bit integer	-2^{31} to $2^{31}-1$
`GrB_INT64`	`int64_t`	64-bit integer	-2^{63} to $2^{63}-1$
`GrB_UINT8`	`uint8_t`	8-bit unsigned integer	0 to 255
`GrB_UINT16`	`uint16_t`	16-bit unsigned integer	0 to $2^{16}-1$
`GrB_UINT32`	`uint32_t`	32-bit unsigned integer	0 to $2^{32}-1$
`GrB_UINT64`	`uint64_t`	64-bit unsigned integer	0 to $2^{64}-1$
`GrB_FP32`	`float`	32-bit IEEE 754	`-Inf` to `+Inf`
`GrB_FP64`	`double`	64-bit IEEE 754	`-Inf` to `+Inf`
`GxB_FC32`	`float complex`	32-bit complex	`-Inf` to `+Inf`
`GxB_FC64`	`double complex`	64-bit complex	`-Inf` to `+Inf`

The ANSI C11 definitions of `float complex` and `double complex` are not always available. The `GraphBLAS.h` header defines them as `GxB_FC32_t` and `GxB_FC64_t`, respectively.

The user application can also define new types based on any `typedef` in the C language whose values are held in a contiguous region of memory of fixed size. For example, a user-defined `GrB_Type` could be created to hold any C `struct` whose content is self-contained. A C `struct` containing pointers might be problematic because GraphBLAS would not know to dereference the pointers to traverse the entire "scalar" entry, but this can be done if the objects referenced by these pointers are not moved. A user-defined complex type with real and imaginary types can be defined, or even a "scalar" type containing a fixed-sized dense matrix (see Section 6.1.1). The possibilities are endless. GraphBLAS can create and operate on sparse matrices and vectors in any of these types, including any user-defined ones. For user-defined types, GraphBLAS simply moves the data around itself (via `memcpy`), and then passes the values back to user-defined functions when it needs to do any computations on the type. The next sections describe the methods for the `GrB_Type` object:

GraphBLAS function	purpose	Section
`GrB_Type_new`	create a user-defined type	6.1.1
`GxB_Type_new`	create a user-defined type, with name and definition	6.1.2
`GrB_Type_wait`	wait for a user-defined type	6.1.3
`GxB_Type_size`	return the size of a type	6.1.4
`GxB_Type_name`	return the name of a type	6.1.5
`GxB_Type_from_name`	return the type from its name	6.1.6
`GrB_Type_free`	free a user-defined type	6.1.7

6.1.1 GrB_Type_new: create a user-defined type

```
GrB_Info GrB_Type_new           // create a new GraphBLAS type
(
    GrB_Type *type,             // handle of user type to create
    size_t sizeof_ctype         // size = sizeof (ctype) of the C type
) ;
```

`GrB_Type_new` creates a new user-defined type. The `type` is a handle, or a pointer to an opaque object. The handle itself must not be `NULL` on input, but the content of the handle can be undefined. On output, the handle contains a pointer to a newly created type. The `ctype` is the type in C that will be used to construct the new GraphBLAS type. It can be either a built-in C type, or defined by a `typedef`. The second parameter should be passed as `sizeof(ctype)`. The only requirement on the C type is that `sizeof(ctype)` is valid in C, and that the type reside in a contiguous block of memory so that it can be moved with `memcpy`. For example, to create a user-defined type called `Complex` for double-precision complex values using the ANSI C11 `double complex` type, the following can be used. A complete example can be found in the `usercomplex.c` and `usercomplex.h` files in the `Demo` folder.

```
#include <math.h>
#include <complex.h>
GrB_Type Complex ;
GrB_Type_new (&Complex, sizeof (double complex)) ;
```

To demonstrate the flexibility of the `GrB_Type`, consider a "scalar" consisting of 4-by-4 floating-point matrix and a string. This type might be useful for the 4-by-4 translation/rotation/scaling matrices that arise in computer graphics, along with a string containing a description or even a regular expression that can be parsed and executed in a user-defined operator. All that is required is a fixed-size type, where `sizeof(ctype)` is a constant.

```
typedef struct
{
    float stuff [4][4] ;
    char whatstuff [64] ;
```

```
}
wildtype ;
GrB_Type WildType ;
GrB_Type_new (&WildType, sizeof (wildtype)) ;
```

With this type a sparse matrix can be created in which each entry consists of a 4-by-4 dense matrix `stuff` and a 64-character string `whatstuff`. GraphBLAS treats this 4-by-4 as a "scalar." Any GraphBLAS method or operation that simply moves data can be used with this type without any further information from the user application. For example, entries of this type can be assigned to and extracted from a matrix or vector, and matrices containing this type can be transposed. A working example (`wildtype.c` in the `Demo` folder) creates matrices and multiplies them with a user-defined semiring with this type.

Performing arithmetic on matrices and vectors with user-defined types requires operators to be defined. Refer to Section 15.5 for more details on these example user-defined types.

6.1.2 GxB_Type_new: create a user-defined type (with name and definition)

```
GrB_Info GxB_Type_new           // create a new named GraphBLAS type
(
    GrB_Type *type,             // handle of user type to create
    size_t sizeof_ctype,        // size = sizeof (ctype) of the C type
    const char *type_name,      // name of the type (max 128 characters)
    const char *type_defn       // typedef for the type (no max length)
) ;
```

`GxB_Type_new` creates a type with a name and definition that are known to GraphBLAS, as strings. The `type_name` is any valid string (max length of 128 characters, including the required null-terminating character) that may appear as the name of a C type created by a C `typedef` statement. It must not contain any white-space characters. For example, to create a type of size 16*4+1 = 65 bytes, with a 4-by-4 dense float array and a 32-bit integer:

```
typedef struct { float x [4][4] ; int color ; } myquaternion ;
GrB_Type MyQtype ;
GxB_Type_new (&MyQtype, sizeof (myquaternion), "myquaternion",
    "typedef struct { float x [4][4] ; int color ; } myquaternion ;") ;
```

The `type_name` and `type_defn` are both null-terminated strings. Currently, `type_defn` is unused, but it will be required for best performance when a JIT is implemented in SuiteSparse:GraphBLAS (both on the CPU and GPU). User defined types created by `GrB_Type_new` will not work with a JIT.

At most `GxB_MAX_NAME_LEN` characters are accessed in `type_name`; characters beyond that limit are silently ignored.

6.1.3 GrB_Type_wait: wait for a type

```
// in SuiteSparse:GraphBLAS v5.x and earlier:
GrB_Info GrB_wait               // wait for a user-defined type
(
    GrB_Type *type              // type to wait for
) ;

// in SuiteSparse:GraphBLAS v6 (v2.0 C API):
GrB_Info GrB_wait               // wait for a user-defined type
(
    GrB_Type type,              // type to wait for
    GrB_WaitMode mode           // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

After creating a user-defined type, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. Currently, SuiteSparse:GraphBLAS currently does nothing except to ensure that `type` is valid.

6.1.4 GxB_Type_size: return the size of a type

```
GrB_Info GxB_Type_size          // determine the size of the type
(
    size_t *size,               // the sizeof the type
    GrB_Type type               // type to determine the sizeof
) ;
```

This function acts just like `sizeof(type)` in the C language. For example `GxB_Type_size (&s, GrB_INT32)` sets `s` to 4, the same as `sizeof(int32_t)`.

6.1.5 GxB_Type_name: return the name of a type

```
GrB_Info GxB_Type_name      // return the name of a GraphBLAS type
(
    char *type_name,        // name of the type (char array of size at least
                            // GxB_MAX_NAME_LEN, owned by the user application).
    const GrB_Type type
) ;
```

Returns the name of a type, as a string. For built-in types, the name is the same as the C type. For example, `GxB_Type_name(type_name,GrB_FP32)` returns the name as `"float"`. The following table lists the names of the 13 built-in types.

Type name	GraphBLAS type
`"bool"`	`GrB_BOOL`
`"int8_t"`	`GrB_INT8`
`"int16_t"`	`GrB_INT16`
`"int32_t"`	`GrB_INT32`
`"int64_t"`	`GrB_INT64`
`"uint8_t"`	`GrB_UINT8`
`"uint16_t"`	`GrB_UINT16`
`"uint32_t"`	`GrB_UINT32`
`"uint64_t"`	`GrB_UINT64`
`"float"`	`GrB_FP32`
`"double"`	`GrB_FP64`
`"float complex"`	`GxB_FC32`
`"double complex"`	`GxB_FC64`

6.1.6 GxB_Type_from_name: **return the type from its name**

```
GrB_Info GxB_Type_from_name     // return the built-in GrB_Type from a name
(
    GrB_Type *type,             // built-in type, or NULL if user-defined
    const char *type_name       // array of size at least GxB_MAX_NAME_LEN
) ;
```

Returns the built-in type from the corresponding name of the type. For example, `GxB_Type_from_name (&type, "bool")` returns `GrB_BOOL`. If the name is from a user-defined type, the `type` is returned as `NULL`. This is not an error condition. The user application must itself do this translation since GraphBLAS does not keep a registry of all user-defined types.

With this function, a user application can manage the translation for both built-in types and its own user-defined types, as in the following example.

```
typedef struct { double x ; char stuff [16] ; } myfirsttype ;
typedef struct { float z [4][4] ; int color ; } myquaternion ;
GrB_Type MyType1, MyQType ;
GxB_Type_new (&MyType1, sizeof (myfirsttype), "myfirsttype",
    "typedef struct { double x ; char stuff [16] ; } myfirsttype ;") ;
GxB_Type_new (&MyQType, sizeof (myquaternion), "myquaternion",
    "typedef struct { float z [4][4] ; int color ; } myquaternion ;") ;

GrB_Matrix A ;
// ... create a matrix A of some built-in or user-defined type

// later on, to query the type of A:
size_t typesize ;
GxB_Type_size (&typesize, type) ;       // works for any type
GrB_Type atype ;
char atype_name [GxB_MAX_NAME_LEN] ;
GxB_Matrix_type_name (atype_name, A) ;
GxB_Type_from_name (&atype, atype_name) ;
if (atype == NULL)
{
    // This is not yet an error.  It means that A has a user-defined type.
    if ((strcmp (atype_name, "myfirsttype")) == 0) atype = MyType1 ;
    else if ((strcmp (atype_name, "myquaternion")) == 0) atype = MyQType ;
    else { ... this is now an error ... the type of A is unknown.  }
}
```

6.1.7 GrB_Type_free: free a user-defined type

```
GrB_Info GrB_free               // free a user-defined type
(
    GrB_Type *type              // handle of user-defined type to free
) ;
```

`GrB_Type_free` frees a user-defined type. Either usage:

```
GrB_Type_free (&type) ;
GrB_free (&type) ;
```

frees the user-defined `type` and sets `type` to `NULL`. It safely does nothing if passed a `NULL` handle, or if `type == NULL` on input.

It is safe to attempt to free a built-in type. SuiteSparse:GraphBLAS silently ignores the request and returns `GrB_SUCCESS`. A user-defined type should not be freed until all operations using the type are completed. SuiteSparse:GraphBLAS attempts to detect this condition but it must query a freed object in its attempt. This is hazardous and not recommended. Operations on such objects whose type has been freed leads to undefined behavior.

It is safe to first free a type, and then a matrix of that type, but after the type is freed the matrix can no longer be used. The only safe thing that can be done with such a matrix is to free it.

The function signature of `GrB_Type_free` uses the generic name `GrB_free`, which can free any GraphBLAS object. See Section 6.15 details. GraphBLAS includes many such generic functions. When describing a specific variation, a function is described with its specific name in this User Guide (such as `GrB_Type_free`). When discussing features applicable to all specific forms, the generic name is used instead (such as `GrB_free`).

6.2 GraphBLAS unary operators: GrB_UnaryOp, $z = f(x)$

A unary operator is a scalar function of the form $z = f(x)$. The domain (type) of z and x need not be the same.

In the notation in the tables below, T is any of the 13 built-in types and is a placeholder for `BOOL`, `INT8`, `UINT8`, ... `FP32`, `FP64`, `FC32`, or `FC64`. For example, `GrB_AINV_INT32` is a unary operator that computes `z=-x` for two values `x` and `z` of type `GrB_INT32`.

The notation R refers to any real type (all but `FC32` and `FC64`), I refers to any integer type (`INT*` and `UINT*`), F refers to any real or complex floating point type (`FP32`, `FP64`, `FC32`, or `FC64`), Z refers to any complex floating point type (`FC32` or `FC64`), and N refers to `INT32` or `INT64`.

The logical negation operator `GrB_LNOT` only works on Boolean types. The `GxB_LNOT_`R functions operate on inputs of type R, implicitly typecasting their input to Boolean and returning result of type R, with a value 1 for true and 0 for false. The operators `GxB_LNOT_BOOL` and `GrB_LNOT` are identical.

Unary operators for all types			
GraphBLAS name	types (domains)	$z = f(x)$	description
`GxB_ONE_`T	$T \to T$	$z = 1$	one
`GrB_IDENTITY_`T	$T \to T$	$z = x$	identity
`GrB_AINV_`T	$T \to T$	$z = -x$	additive inverse
`GrB_MINV_`T	$T \to T$	$z = 1/x$	multiplicative inverse

Unary operators for real and integer types			
GraphBLAS name	types (domains)	$z = f(x)$	description
`GrB_ABS_`T	$R \to R$	$z = \|x\|$	absolute value
`GrB_LNOT`	`bool` $\to$ `bool`	$z = \neg x$	logical negation
`GxB_LNOT_`R	$R \to R$	$z = \neg(x \neq 0)$	logical negation
`GrB_BNOT_`I	$I \to I$	$z = \neg x$	bitwise negation

Positional unary operators for any type (including user-defined)			
GraphBLAS name	types (domains)	$z = f(a_{ij})$	description
`GxB_POSITIONI_`N	$\to N$	$z = i$	row index (0-based)
`GxB_POSITIONI1_`N	$\to N$	$z = i + 1$	row index (1-based)
`GxB_POSITIONJ_`N	$\to N$	$z = j$	column index (0-based)
`GxB_POSITIONJ1_`N	$\to N$	$z = j + 1$	column index (1-based)

Unary operators for floating-point types (real and complex)			
GraphBLAS name	types (domains)	$z = f(x)$	description
`GxB_SQRT_`F	$F \to F$	$z = \sqrt{(x)}$	square root
`GxB_LOG_`F	$F \to F$	$z = \log_e(x)$	natural logarithm
`GxB_EXP_`F	$F \to F$	$z = e^x$	natural exponent
`GxB_LOG10_`F	$F \to F$	$z = \log_{10}(x)$	base-10 logarithm
`GxB_LOG2_`F	$F \to F$	$z = \log_2(x)$	base-2 logarithm
`GxB_EXP2_`F	$F \to F$	$z = 2^x$	base-2 exponent
`GxB_EXPM1_`F	$F \to F$	$z = e^x - 1$	natural exponent - 1
`GxB_LOG1P_`F	$F \to F$	$z = \log(x + 1)$	natural log of $x + 1$
`GxB_SIN_`F	$F \to F$	$z = \sin(x)$	sine
`GxB_COS_`F	$F \to F$	$z = \cos(x)$	cosine
`GxB_TAN_`F	$F \to F$	$z = \tan(x)$	tangent
`GxB_ASIN_`F	$F \to F$	$z = \sin^{-1}(x)$	inverse sine
`GxB_ACOS_`F	$F \to F$	$z = \cos^{-1}(x)$	inverse cosine
`GxB_ATAN_`F	$F \to F$	$z = \tan^{-1}(x)$	inverse tangent
`GxB_SINH_`F	$F \to F$	$z = \sinh(x)$	hyperbolic sine
`GxB_COSH_`F	$F \to F$	$z = \cosh(x)$	hyperbolic cosine
`GxB_TANH_`F	$F \to F$	$z = \tanh(x)$	hyperbolic tangent
`GxB_ASINH_`F	$F \to F$	$z = \sinh^{-1}(x)$	inverse hyperbolic sine
`GxB_ACOSH_`F	$F \to F$	$z = \cosh^{-1}(x)$	inverse hyperbolic cosine
`GxB_ATANH_`F	$F \to F$	$z = \tanh^{-1}(x)$	inverse hyperbolic tangent
`GxB_SIGNUM_`F	$F \to F$	$z = \text{sgn}(x)$	sign, or signum function
`GxB_CEIL_`F	$F \to F$	$z = \lceil x \rceil$	ceiling function
`GxB_FLOOR_`F	$F \to F$	$z = \lfloor x \rfloor$	floor function
`GxB_ROUND_`F	$F \to F$	$z = \text{round}(x)$	round to nearest
`GxB_TRUNC_`F	$F \to F$	$z = \text{trunc}(x)$	round towards zero
`GxB_LGAMMA_`F	$F \to F$	$z = \log(\lvert\Gamma(x)\rvert)$	log of gamma function
`GxB_TGAMMA_`F	$F \to F$	$z = \Gamma(x)$	gamma function
`GxB_ERF_`F	$F \to F$	$z = \text{erf}(x)$	error function
`GxB_ERFC_`F	$F \to F$	$z = \text{erfc}(x)$	complimentary error function
`GxB_FREXPX_`F	$F \to F$	$z = \text{frexpx}(x)$	normalized fraction
`GxB_FREXPE_`F	$F \to F$	$z = \text{frexpe}(x)$	normalized exponent
`GxB_ISINF_`F	$F \to$ `bool`	$z = \text{isinf}(x)$	true if $\pm\infty$
`GxB_ISNAN_`F	$F \to$ `bool`	$z = \text{isnan}(x)$	true if `NaN`
`GxB_ISFINITE_`F	$F \to$ `bool`	$z = \text{isfinite}(x)$	true if finite

Unary operators for complex types			
GraphBLAS name	types (domains)	$z = f(x)$	description
`GxB_CONJ_`Z	$Z \to Z$	$z = \overline{x}$	complex conjugate
`GxB_ABS_`Z	$Z \to F$	$z = \lvert x \rvert$	absolute value
`GxB_CREAL_`Z	$Z \to F$	$z = \text{real}(x)$	real part
`GxB_CIMAG_`Z	$Z \to F$	$z = \text{imag}(x)$	imaginary part
`GxB_CARG_`Z	$Z \to F$	$z = \text{carg}(x)$	angle

A positional unary operator return the row or column index of an entry. For a matrix $z = f(a_{ij})$ returns $z = i$ or $z = j$, or $+1$ for 1-based indices. The latter is useful in the Octave/MATLAB interface, where row and column indices are 1-based. When applied to a vector, j is always zero, and i is the index in the vector. Positional unary operators come in two types: `INT32` and `INT64`, which is the type of the output, z. The functions are agnostic to the type of their inputs; they only depend on the position of the entries, not their values. User-defined positional operators cannot be defined by `GrB_UnaryOp_new`.

`GxB_FREXPX` and `GxB_FREXPE` return the mantissa and exponent, respectively, from the ANSI C11 `frexp` function. The exponent is returned as a floating-point value, not an integer.

The operators `GxB_EXPM1_FC*` and `GxB_LOG1P_FC*` for complex types are currently not accurate. They will be revised in a future version.

The functions `casin`, `casinf`, `casinh`, and `casinhf` provided by Microsoft Visual Studio for computing $\sin^{-1}(x)$ and $\sinh^{-1}(x)$ when x is complex do not compute the correct result. Thus, the unary operators `GxB_ASIN_FC32`, `GxB_ASIN_FC64` `GxB_ASINH_FC32`, and `GxB_ASINH_FC64` do not work properly if the MS Visual Studio compiler is used. These functions work properly if the gcc, icc, or clang compilers are used on Linux or MacOS.

Integer division by zero normally terminates an application, but this is avoided in SuiteSparse:GraphBLAS. For details, see the binary `GrB_DIV_`T operators.

SPEC: The definition of integer division by zero is an extension to the specification.

The next sections define the following methods for the `GrB_UnaryOp` object:

GraphBLAS function	purpose	Section
`GrB_UnaryOp_new`	create a user-defined unary operator	6.2.1
`GxB_UnaryOp_new`	create a named user-defined unary operator	6.2.2
`GrB_UnaryOp_wait`	wait for a user-defined unary operator	6.2.3
`GxB_UnaryOp_ztype_name`	return the name of the type of the output z for $z = f(x)$	6.2.4
`GxB_UnaryOp_xtype_name`	return the name of the type of the input x for $z = f(x)$	6.2.5
`GrB_UnaryOp_free`	free a user-defined unary operator	6.2.6

6.2.1 GrB_UnaryOp_new: create a user-defined unary operator

```
GrB_Info GrB_UnaryOp_new            // create a new user-defined unary operator
(
    GrB_UnaryOp *unaryop,           // handle for the new unary operator
    void *function,                 // pointer to the unary function
    GrB_Type ztype,                 // type of output z
    GrB_Type xtype                  // type of input x
) ;
```

`GrB_UnaryOp_new` creates a new unary operator. The new operator is returned in the `unaryop` handle, which must not be `NULL` on input. On output, its contents contains a pointer to the new unary operator.

The two types `xtype` and `ztype` are the GraphBLAS types of the input x and output z of the user-defined function $z = f(x)$. These types may be built-in types or user-defined types, in any combination. The two types need not be the same, but they must be previously defined before passing them to `GrB_UnaryOp_new`.

The `function` argument to `GrB_UnaryOp_new` is a pointer to a user-defined function with the following signature:

```
void (*f) (void *z, const void *x) ;
```

When the function `f` is called, the arguments `z` and `x` are passed as `(void *)` pointers, but they will be pointers to values of the correct type, defined by `ztype` and `xtype`, respectively, when the operator was created.

NOTE: The pointers passed to a user-defined operator may not be unique. That is, the user function may be called with multiple pointers that point to the same space, such as when `z=f(z,y)` is to be computed by a binary operator, or `z=f(z)` for a unary operator. Any parameters passed to the user-callable function may be aliased to each other.

6.2.2 GxB_UnaryOp_new: create a named user-defined unary operator

```
GrB_Info GxB_UnaryOp_new            // create a new user-defined unary operator
(
    GrB_UnaryOp *unaryop,           // handle for the new unary operator
    GxB_unary_function function,    // pointer to the unary function
    GrB_Type ztype,                 // type of output z
    GrB_Type xtype,                 // type of input x
    const char *unop_name,          // name of the user function
    const char *unop_defn           // definition of the user function
) ;
```

Creates a named `GrB_UnaryOp`. Only the first 127 characters of `unop_name` are used. The `unop_defn` is a string containing the entire function itself. For example:

```
void square (double *z, double *x) { (*z) = (*x) * (*x) ; } ;
...
GrB_Type Square ;
GxB_UnaryOp_new (&Square, square, GrB_FP64, GrB_FP64, "square",
    "void square (double *z, double *x) { (*z) = (*x) * (*x) ; } ;") ;
```

Currently, only the `unop_name` is used, but future versions will rely on the `unop_defn` when employing a JIT for better performance.

6.2.3 GrB_UnaryOp_wait: wait for a unary operator

```
// in SuiteSparse:GraphBLAS v5.x and earlier:
GrB_Info GrB_wait               // wait for a user-defined unary operator
(
    GrB_UnaryOp *unaryop        // unary operator to wait for
) ;

// in SuiteSparse:GraphBLAS v6 (v2.0 C API):
GrB_Info GrB_wait               // wait for a user-defined unary operator
(
    GrB_UnaryOp unaryop,        // unary operator to wait for
    GrB_WaitMode mode           // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

After creating a user-defined unary operator, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. Currently, SuiteSparse:GraphBLAS currently does nothing except to ensure that the `unaryop` is valid.

6.2.4 GxB_UnaryOp_ztype_name: return the name of the type of z

```
GrB_Info GxB_UnaryOp_ztype_name     // return the type_name of z
(
    char *type_name,                // user array of size GxB_MAX_NAME_LEN
    const GrB_UnaryOp unaryop       // unary operator
) ;
```

`GxB_UnaryOp_ztype_name` returns the name of the `ztype` of the unary operator, which is the type of z in the function $z = f(x)$.

6.2.5 GxB_UnaryOp_xtype_name: return the name of the type of x

```
GrB_Info GxB_UnaryOp_xtype_name     // return the type_name of x
(
    char *type_name,                // user array of size GxB_MAX_NAME_LEN
    const GrB_UnaryOp unaryop       // unary operator
) ;
```

`GxB_UnaryOp_xtype_name` returns the name of the `xtype` of the unary operator, which is the type of x in the function $z = f(x)$.

6.2.6 GrB_UnaryOp_free: free a user-defined unary operator

```
GrB_Info GrB_free                   // free a user-created unary operator
(
    GrB_UnaryOp *unaryop            // handle of unary operator to free
) ;
```

`GrB_UnaryOp_free` frees a user-defined unary operator. Either usage:

```
GrB_UnaryOp_free (&unaryop) ;
GrB_free (&unaryop) ;
```

frees the `unaryop` and sets `unaryop` to `NULL`. It safely does nothing if passed a `NULL` handle, or if `unaryop == NULL` on input. It does nothing at all if passed a built-in unary operator.

6.3 GraphBLAS binary operators: GrB_BinaryOp, $z = f(x, y)$

A binary operator is a scalar function of the form $z = f(x, y)$. The types of z, x, and y need not be the same. The built-in binary operators are listed in the tables below. The notation T refers to any of the 13 built-in types, but two of those types are SuiteSparse extensions (`GxB_FC32` and `GxB_FC64`). For those types, the operator name always starts with `GxB`, not `GrB`).

The six `GxB_IS*` comparators and the `GxB_*` logical operators all return a result one for true and zero for false, in the same domain T or R as their inputs. These six comparators are useful as "multiply" operators for creating semirings with non-Boolean monoids.

Binary operators for all 13 types			
GraphBLAS name	types (domains)	$z = f(x, y)$	description
`GrB_FIRST_`T	$T \times T \to T$	$z = x$	first argument
`GrB_SECOND_`T	$T \times T \to T$	$z = y$	second argument
`GxB_ANY_`T	$T \times T \to T$	$z = x$ or y	pick x or y arbitrarily
`GrB_ONEB_`T	$T \times T \to T$	$z = 1$	one
`GxB_PAIR_`T	$T \times T \to T$	$z = 1$	one (historical)
`GrB_PLUS_`T	$T \times T \to T$	$z = x + y$	addition
`GrB_MINUS_`T	$T \times T \to T$	$z = x - y$	subtraction
`GxB_RMINUS_`T	$T \times T \to T$	$z = y - x$	reverse subtraction
`GrB_TIMES_`T	$T \times T \to T$	$z = xy$	multiplication
`GrB_DIV_`T	$T \times T \to T$	$z = x/y$	division
`GxB_RDIV_`T	$T \times T \to T$	$z = y/x$	reverse division
`GxB_POW_`T	$T \times T \to T$	$z = x^y$	power
`GxB_ISEQ_`T	$T \times T \to T$	$z = (x == y)$	equal
`GxB_ISNE_`T	$T \times T \to T$	$z = (x \neq y)$	not equal

The `GxB_POW_*` operators for real types do not return a complex result, and thus $z = f(x, y) = x^y$ is undefined if x is negative and y is not an integer. To compute a complex result, use `GxB_POW_FC32` or `GxB_POW_FC64`.

Operators that require the domain to be ordered (`MIN`, `MAX`, less-than, greater-than, and so on) are not defined for complex types. These are listed in the following table:

Binary operators for all non-complex types			
GraphBLAS name	types (domains)	$z = f(x, y)$	description
`GrB_MIN_`R	$R \times R \to R$	$z = \min(x, y)$	minimum
`GrB_MAX_`R	$R \times R \to R$	$z = \max(x, y)$	maximum
`GxB_ISGT_`R	$R \times R \to R$	$z = (x > y)$	greater than
`GxB_ISLT_`R	$R \times R \to R$	$z = (x < y)$	less than
`GxB_ISGE_`R	$R \times R \to R$	$z = (x \geq y)$	greater than or equal
`GxB_ISLE_`R	$R \times R \to R$	$z = (x \leq y)$	less than or equal
`GxB_LOR_`R	$R \times R \to R$	$z = (x \neq 0) \vee (y \neq 0)$	logical OR
`GxB_LAND_`R	$R \times R \to R$	$z = (x \neq 0) \wedge (y \neq 0)$	logical AND
`GxB_LXOR_`R	$R \times R \to R$	$z = (x \neq 0) \veebar (y \neq 0)$	logical XOR

Another set of six kinds of built-in comparators have the form $T \times T \to$`bool`. Note that when T is `bool`, the six operators give the same results as the six `GxB_IS*_BOOL` operators in the table above. These six comparators are useful as "multiply" operators for creating semirings with Boolean monoids.

Binary comparators for all 13 types			
GraphBLAS name	types (domains)	$z = f(x, y)$	description
`GrB_EQ_`T	$T \times T \to$`bool`	$z = (x == y)$	equal
`GrB_NE_`T	$T \times T \to$`bool`	$z = (x \neq y)$	not equal

Binary comparators for non-complex types			
GraphBLAS name	types (domains)	$z = f(x, y)$	description
`GrB_GT_`R	$R \times R \to$`bool`	$z = (x > y)$	greater than
`GrB_LT_`R	$R \times R \to$`bool`	$z = (x < y)$	less than
`GrB_GE_`R	$R \times R \to$`bool`	$z = (x \geq y)$	greater than or equal
`GrB_LE_`R	$R \times R \to$`bool`	$z = (x \leq y)$	less than or equal

GraphBLAS has four built-in binary operators that operate purely in the Boolean domain. The first three are identical to the `GxB_L*_BOOL` operators described above, just with a shorter name. The `GrB_LXNOR` operator is the same as `GrB_EQ_BOOL`.

Binary operators for the boolean type only			
GraphBLAS name	types (domains)	$z = f(x, y)$	description
`GrB_LOR`	`bool` $\times$ `bool` $\to$ `bool`	$z = x \vee y$	logical OR
`GrB_LAND`	`bool` $\times$ `bool` $\to$ `bool`	$z = x \wedge y$	logical AND
`GrB_LXOR`	`bool` $\times$ `bool` $\to$ `bool`	$z = x \veebar y$	logical XOR
`GrB_LXNOR`	`bool` $\times$ `bool` $\to$ `bool`	$z = \neg(x \veebar y)$	logical XNOR

The following operators are defined for real floating-point types only (`GrB_FP32` and `GrB_FP64`). They are identical to the ANSI C11 functions of the same name. The last one in the table constructs the corresponding complex type.

Binary operators for the real floating-point types only			
GraphBLAS name	types (domains)	$z = f(x, y)$	description
`GxB_ATAN2_`F	$F \times F \to F$	$z = \tan^{-1}(y/x)$	4-quadrant arc tangent
`GxB_HYPOT_`F	$F \times F \to F$	$z = \sqrt{x^2 + y^2}$	hypotenuse
`GxB_FMOD_`F	$F \times F \to F$		ANSI C11 `fmod`
`GxB_REMAINDER_`F	$F \times F \to F$		ANSI C11 `remainder`
`GxB_LDEXP_`F	$F \times F \to F$		ANSI C11 `ldexp`
`GxB_COPYSIGN_`F	$F \times F \to F$		ANSI C11 `copysign`
`GxB_CMPLX_`F	$F \times F \to Z$	$z = x + y \times i$	complex from real & imag

Eight bitwise operators are predefined for signed and unsigned integers.

Binary operators for signed and unsigned integers			
GraphBLAS name	types (domains)	$z = f(x, y)$	description
`GrB_BOR_`I	$I \times I \to I$	`z=x|y`	bitwise logical OR
`GrB_BAND_`I	$I \times I \to I$	`z=x&y`	bitwise logical AND
`GrB_BXOR_`I	$I \times I \to I$	`z=x^y`	bitwise logical XOR
`GrB_BXNOR_`I	$I \times I \to I$	`z=~(x^y)`	bitwise logical XNOR
`GxB_BGET_`I	$I \times I \to I$		get bit y of x
`GxB_BSET_`I	$I \times I \to I$		set bit y of x
`GxB_BCLR_`I	$I \times I \to I$		clear bit y of x
`GxB_BSHIFT_`I	$I \times$`int8`$\to I$		bit shift

There are two sets of built-in comparators in SuiteSparse:GraphBLAS, but they are not redundant. They are identical except for the type (domain) of their output, z. The `GrB_EQ_`T and related operators compare their inputs of type T and produce a Boolean result of true or false. The `GxB_ISEQ_`T and related operators compute the same thing and produce a result with same type T as their input operands, returning one for true or zero for false. The `IS*` comparators are useful when combining comparators with other non-Boolean operators. For example, a `PLUS-ISEQ` semiring counts how many terms are true. With this semiring, matrix multiplication $\mathbf{C} = \mathbf{AB}$ for two weighted undirected graphs $\mathbf{A}$ and $\mathbf{B}$ computes c_{ij} as the number of edges node i and j have in common that have identical edge weights. Since the output type of the "multiplier" operator in a semiring must match the type of its monoid, the Boolean `EQ` cannot be combined with a non-Boolean `PLUS` monoid to perform this operation.

Likewise, SuiteSparse:GraphBLAS has two sets of logical OR, AND, and XOR operators. Without the `_`T suffix, the three operators `GrB_LOR`, `GrB_LAND`, and `GrB_LXOR` operate purely in the Boolean domain, where all input and output types are `GrB_BOOL`. The second set (`GxB_LOR_`T `GxB_LAND_`T and `GxB_LXOR_`T) provides Boolean operators to all 11 real domains, implicitly typecasting their inputs from type T to Boolean and returning a value of type T that is 1 for true or zero for false. The set of `GxB_L*_`T operators are useful since they can be combined with non-Boolean monoids in a semiring.

Floating-point operations follow the IEEE 754 standard. Thus, computing $x/0$ for a floating-point x results in `+Inf` if x is positive, `-Inf` if x is negative, and `NaN` if x

is zero. The application is not terminated. However, integer division by zero normally terminates an application. SuiteSparse:GraphBLAS avoids this by adopting the same rules as MATLAB, which are analogous to how the IEEE standard handles floating-point division by zero. For integers, when x is positive, $x/0$ is the largest positive integer, for negative x it is the minimum integer, and $0/0$ results in zero. For example, for an integer x of type `GrB_INT32`, $1/0$ is $2^{31} - 1$ and (-1)/0 is -2^{31}. Refer to Section 6.1 for a list of integer ranges.

Eight positional operators are predefined. They differ when used in a semiring and when used in `GrB_eWise*` and `GrB_apply`. Positional operators cannot be used in `GrB_build`, nor can they be used as the `accum` operator for any operation.

The positional binary operators do not depend on the type or numerical value of their inputs, just their position in a matrix or vector. For a vector, j is always 0, and i is the index into the vector. There are two types N available: `INT32` and `INT64`, which is the type of the output z. User-defined positional operators cannot be defined by `GrB_BinaryOp_new`.

Positional binary operators for any type (including user-defined) when used as a multiplicative operator in a semiring			
GraphBLAS name	types (domains)	$z = f(a_{ik}, b_{kj})$	description
`GxB_FIRSTI_`N	$\rightarrow N$	$z = i$	row index of a_{ik} (0-based)
`GxB_FIRSTI1_`N	$\rightarrow N$	$z = i + 1$	row index of a_{ik} (1-based)
`GxB_FIRSTJ_`N	$\rightarrow N$	$z = k$	column index of a_{ik} (0-based)
`GxB_FIRSTJ1_`N	$\rightarrow N$	$z = k + 1$	column index of a_{ik} (1-based)
`GxB_SECONDI_`N	$\rightarrow N$	$z = k$	row index of b_{kj} (0-based)
`GxB_SECONDI1_`N	$\rightarrow N$	$z = k + 1$	row index of b_{kj} (1-based)
`GxB_SECONDJ_`N	$\rightarrow N$	$z = j$	column index of b_{kj} (0-based)
`GxB_SECONDJ1_`N	$\rightarrow N$	$z = j + 1$	column index of b_{kj} (1-based)

Positional binary operators for any type (including user-defined) when used in all other methods			
GraphBLAS name	types (domains)	$z = f(a_{ij}, b_{ij})$	description
`GxB_FIRSTI_`N	$\rightarrow N$	$z = i$	row index of a_{ij} (0-based)
`GxB_FIRSTI1_`N	$\rightarrow N$	$z = i + 1$	row index of a_{ij} (1-based)
`GxB_FIRSTJ_`N	$\rightarrow N$	$z = j$	column index of a_{ij} (0-based)
`GxB_FIRSTJ1_`N	$\rightarrow N$	$z = j + 1$	column index of a_{ij} (1-based)
`GxB_SECONDI_`N	$\rightarrow N$	$z = i$	row index of b_{ij} (0-based)
`GxB_SECONDI1_`N	$\rightarrow N$	$z = i + 1$	row index of b_{ij} (1-based)
`GxB_SECONDJ_`N	$\rightarrow N$	$z = j$	column index of b_{ij} (0-based)
`GxB_SECONDJ1_`N	$\rightarrow N$	$z = j + 1$	column index of b_{ij} (1-based)

Finally, one special binary operator can only be used as input to `GrB_Matrix_build` or `GrB_Vector_build`: the `GxB_IGNORE_DUP` operator. If `dup` is `NULL`, any duplicates in the `GrB*build` methods result in an error. If `dup` is the special binary operator `GxB_IGNORE_DUP`, then any duplicates are ignored. If duplicates appear, the last one in the list of tuples is taken and the prior ones ignored. This is not an error.

The next sections define the following methods for the `GrB_BinaryOp` object:

GraphBLAS function	purpose	Section
`GrB_BinaryOp_new`	create a user-defined binary operator	6.3.1
`GxB_BinaryOp_new`	create a named user-defined binary operator	6.3.2
`GrB_BinaryOp_wait`	wait for a user-defined binary operator	6.3.3
`GxB_BinaryOp_ztype_name`	return the type of the output z for $z = f(x, y)$	6.3.4
`GxB_BinaryOp_xtype_name`	return the type of the input x for $z = f(x, y)$	6.3.5
`GxB_BinaryOp_ytype_name`	return the type of the input y for $z = f(x, y)$	6.3.6
`GrB_BinaryOp_free`	free a user-defined binary operator	6.3.7

6.3.1 GrB_BinaryOp_new: create a user-defined binary operator

```
GrB_Info GrB_BinaryOp_new
(
    GrB_BinaryOp *binaryop,         // handle for the new binary operator
    void *function,                 // pointer to the binary function
    GrB_Type ztype,                 // type of output z
    GrB_Type xtype,                 // type of input x
    GrB_Type ytype                  // type of input y
) ;
```

`GrB_BinaryOp_new` creates a new binary operator. The new operator is returned in the `binaryop` handle, which must not be `NULL` on input. On output, its contents contains a pointer to the new binary operator.

The three types `xtype`, `ytype`, and `ztype` are the GraphBLAS types of the inputs x and y, and output z of the user-defined function $z = f(x, y)$. These types may be built-in types or user-defined types, in any combination. The three types need not be the same, but they must be previously defined before passing them to `GrB_BinaryOp_new`.

The final argument to `GrB_BinaryOp_new` is a pointer to a user-defined function with the following signature:

```
void (*f) (void *z, const void *x, const void *y) ;
```

When the function `f` is called, the arguments `z`, `x`, and `y` are passed as `(void *)` pointers, but they will be pointers to values of the correct type, defined by `ztype`, `xtype`, and `ytype`, respectively, when the operator was created.

NOTE: SuiteSparse:GraphBLAS may call the function with the pointers `z` and `x` equal to one another, in which case `z=f(z,y)` should be computed. Future versions may use additional pointer aliasing.

6.3.2 GxB_BinaryOp_new: create a named user-defined binary operator

```
GrB_Info GxB_BinaryOp_new
(
    GrB_BinaryOp *op,               // handle for the new binary operator
    GxB_binary_function function,   // pointer to the binary function
    GrB_Type ztype,                 // type of output z
    GrB_Type xtype,                 // type of input x
    GrB_Type ytype,                 // type of input y
    const char *binop_name,         // name of the user function
    const char *binop_defn          // definition of the user function
) ;
```

Creates a named `GrB_BinaryOp`. Only the first 127 characters of `binop_name` are used. The `binop_defn` is a string containing the entire function itself. For example:

```
void absdiff (double *z, double *x, double *y) { (*z) = fabs ((*x) - (*y)) ; } ;
...
GrB_Type AbsDiff ;
GxB_BinaryOp_new (&AbsDiff, absdiff, GrB_FP64, GrB_FP64, GrB_FP64, "absdiff",
  "void absdiff (double *z, double *x, double *y) { (*z) = fabs ((*x) - (*y)) ; }") ;
```

Currently, only the `binop_name` is used, but future versions will rely on the `binop_defn` when employing a JIT for better performance.

6.3.3 GrB_BinaryOp_wait: wait for a binary operator

```
// in SuiteSparse:GraphBLAS v5.x and earlier:
GrB_Info GrB_wait               // wait for a user-defined binary operator
(
    GrB_BinaryOp *binaryop      // binary operator to wait for
) ;

// in SuiteSparse:GraphBLAS v6 (v2.0 C API):
GrB_Info GrB_wait               // wait for a user-defined binary operator
(
    GrB_BinaryOp binaryop,      // binary operator to wait for
    GrB_WaitMode mode           // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

After creating a user-defined binary operator, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. Currently, SuiteSparse:GraphBLAS currently does nothing for except to ensure that the `binaryop` is valid.

6.3.4 GxB_BinaryOp_ztype_name: return the name of the type of z

```
GrB_Info GxB_BinaryOp_ztype_name    // return the type_name of z
(
    char *type_name,                // user array of size GxB_MAX_NAME_LEN
    const GrB_BinaryOp binaryop     // binary operator to query
) ;
```

`GxB_BinaryOp_ztype_name` returns name of the `ztype` of the binary operator, which is the type of z in the function $z = f(x, y)$.

6.3.5 GxB_BinaryOp_xtype_name: return the name of the type of x

```
GrB_Info GxB_BinaryOp_xtype_name    // return the type_name of x
(
    char *type_name,                // user array of size GxB_MAX_NAME_LEN
    const GrB_BinaryOp binaryop     // binary operator to query
) ;
```

`GxB_BinaryOp_xtype_name` returns name of the `xtype` of the binary operator, which is the type of x in the function $z = f(x, y)$.

6.3.6 GxB_BinaryOp_ytype_name: return the name of the type of y

```
GrB_Info GxB_BinaryOp_ytype_name    // return the type_name of y
(
    char *type_name,                // user array of size GxB_MAX_NAME_LEN
    const GrB_BinaryOp binaryop     // binary operator to query
) ;
```

`GxB_BinaryOp_ytype_name` returns name of the `ytype` of the binary operator, which is the type of y in the function $z = f(x, y)$.

6.3.7 GrB_BinaryOp_free: free a user-defined binary operator

```
GrB_Info GrB_free                   // free a user-created binary operator
(
    GrB_BinaryOp *binaryop          // handle of binary operator to free
) ;
```

`GrB_BinaryOp_free` frees a user-defined binary operator. Either usage:

```
GrB_BinaryOp_free (&op) ;
GrB_free (&op) ;
```

frees the `op` and sets `op` to `NULL`. It safely does nothing if passed a `NULL` handle, or if `op == NULL` on input. It does nothing at all if passed a built-in binary operator.

6.3.8 ANY and PAIR (ONEB) operators

The `GxB_PAIR` operator (also called `GrB_ONEB`) is simple to describe: just $f(x, y) = 1$. It is called the `PAIR` operator since it returns 1 in a semiring when a pair of entries a_{ik} and b_{kj} is found in the matrix multiply. This operator is simple yet very useful. It allows purely structural computations to be performed on matrices of any type, without having to typecast them to Boolean with all values being true. Typecasting need not be performed on the inputs to the `PAIR` operator, and the `PAIR` operator does not need to access the values of the matrix. This cuts memory accesses, so it is a very fast operator to use.

The `GxB_PAIR_T` operator is a SuiteSparse:GraphBLAS extension. It has since been added to the v2.0 C API Specification as `GrB_ONEB_T`. They are identical, but the latter name should be used for compatibility with other GraphBLAS libraries.

The `ANY` operator is very unusual, but very powerful. It is the function $f_{\text{any}}(x, y) = x$, or y, where GraphBLAS has to freedom to select either x, or y, at its own discretion. Do not confuse the `ANY` operator with the `any` function in Octave/MATLAB, which computes a reduction using the logical OR operator.

The `ANY` function is associative and commutative, and can thus serve as an operator for a monoid. The selection of x are y is not randomized. Instead, SuiteSparse:GraphBLAS

uses this freedom to compute as fast a result as possible. When used as the monoid in a dot product,

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

for example, the computation can terminate as soon as any matching pair of entries is found. When used in a parallel saxpy-style computation, the `ANY` operator allows for a relaxed form of synchronization to be used, resulting in a fast benign race condition.

Because of this benign race condition, the result of the `ANY` monoid can be non-deterministic, unless it is coupled with the `PAIR` multiplicative operator. In this case, the `ANY_PAIR` semiring will return a deterministic result, since $f_{\text{any}}(1, 1)$ is always 1.

When paired with a different operator, the results are non-deterministic. This gives a powerful method when computing results for which any value selected by the `ANY` operator is valid. One such example is the breadth-first-search tree. Suppose node j is at level v, and there are multiple nodes i at level $v - 1$ for which the edge (i, j) exists in the graph. Any of these nodes i can serve as a valid parent in the BFS tree. Using the `ANY` operator, GraphBLAS can quickly compute a valid BFS tree; if it used again on the same inputs, it might return a different, yet still valid, BFS tree, due to the non-deterministic nature of intra-thread synchronization.

6.4 GraphBLAS IndexUnaryOp operators: GrB_IndexUnaryOp

An index-unary operator is a scalar function of the form $z = f(a_{ij}, i, j, y)$ that is applied to the entries a_{ij} of an m-by-n matrix. It can be used in `GrB_apply` (Section 10.12) or in `GrB_select` (Section 10.13) to select entries from a matrix or vector.

The signature of the index-unary function `f` is as follows:

```
void f
(
    void *z,            // output value z, of type ztype
    const void *x,      // input value x of type xtype; value of v(i) or A(i,j)
    GrB_Index i,        // row index of A(i,j)
    GrB_Index j,        // column index of A(i,j), or zero for v(i)
    const void *y       // input scalar y
) ;
```

The following built-in operators are available. Operators that do not depend on the value of `A(i,j)` can be used on any matrix or vector, including those of user-defined type. In the table, `y` is a scalar whose type matches the suffix of the operator. The `VALUEEQ` and `VALUENE` operators are defined for any built-in type. The other `VALUE` operators are defined only for real (not complex) built-in types. Any index computations are done in `int64_t` arithmetic; the result is typecasted to `int32_t` for the `*INDEX_INT32` operators.

GraphBLAS name	Octave/MATLAB analog	description
`GrB_ROWINDEX_INT32`	`z=i+y`	row index of `A(i,j)`, as int32
`GrB_ROWINDEX_INT64`	`z=i+y`	row index of `A(i,j)`, as int64
`GrB_COLINDEX_INT32`	`z=j+y`	column index of `A(i,j)`, as int32
`GrB_COLINDEX_INT64`	`z=j+y`	column index of `A(i,j)`, as int64
`GrB_DIAGINDEX_INT32`	`z=j-(i+y)`	column diagonal index of `A(i,j)`, as int32
`GrB_DIAGINDEX_INT64`	`z=j-(i+y)`	column diagonal index of `A(i,j)`, as int64
`GrB_TRIL`	`z=(j<=(i+y))`	true for entries on or below the `y`th diagonal
`GrB_TRIU`	`z=(j>=(i+y))`	true for entries on or above the `y`th diagonal
`GrB_DIAG`	`z=(j==(i+y))`	true for entries on the `y`th diagonal
`GrB_OFFDIAG`	`z=(j!=(i+y))`	true for entries not on the `y`th diagonal
`GrB_COLLE`	`z=(j<=y)`	true for entries in columns 0 to `y`
`GrB_COLGT`	`z=(j>y)`	true for entries in columns `y+1` and above
`GrB_ROWLE`	`z=(i<=y)`	true for entries in rows 0 to `y`
`GrB_ROWGT`	`z=(i>y)`	true for entries in rows `y+1` and above
`GrB_VALUENE_T`	`z=(aij!=y)`	true if `A(i,j)` is not equal to `y`
`GrB_VALUEEQ_T`	`z=(aij==y)`	true if `A(i,j)` is equal to `y`
`GrB_VALUEGT_T`	`z=(aij>y)`	true if `A(i,j)` is greater than `y`
`GrB_VALUEGE_T`	`z=(aij>=y)`	true if `A(i,j)` is greater than or equal to `y`
`GrB_VALUELT_T`	`z=(aij<y)`	true if `A(i,j)` is less than `y`
`GrB_VALUELE_T`	`z=(aij<=y)`	true if `A(i,j)` is less than or equal to `y`

The following methods operate on the `GrB_IndexUnaryOp` object:

GraphBLAS function	purpose	Section
`GrB_IndexUnaryOp_new`	create a user-defined index-unary operator	6.4.1
`GxB_IndexUnaryOp_new`	create a named user-defined index-unary operator	6.4.2
`GrB_IndexUnaryOp_wait`	wait for a user-defined index-unary operator	6.4.3
`GrB_IndexUnaryOp_ztype_name`	return the type of the output z	6.4.4
`GrB_IndexUnaryOp_xtype_name`	return the type of the input x	6.4.5
`GrB_IndexUnaryOp_ytype_name`	return the type of the scalar y	6.4.6
`GrB_IndexUnaryOp_free`	free a user-defined index-unary operator	6.4.7

6.4.1 GrB_IndexUnaryOp_new: create a user-defined index-unary operator

```
GrB_Info GrB_IndexUnaryOp_new       // create a new user-defined IndexUnary op
(
    GrB_IndexUnaryOp *op,           // handle for the new IndexUnary operator
    void *function,                 // pointer to IndexUnary function
    GrB_Type ztype,                 // type of output z
    GrB_Type xtype,                 // type of input x (the A(i,j) entry)
    GrB_Type ytype                  // type of scalar input y
) ;
```

`GrB_IndexUnaryOp_new` creates a new index-unary operator. The new operator is returned in the `op` handle, which must not be `NULL` on input. On output, its contents contains a pointer to the new index-unary operator.

The `function` argument to `GrB_IndexUnaryOp_new` is a pointer to a user-defined function whose signature is given at the beginning of Section 6.4. Given the properties of an entry a_{ij} in a matrix, the `function` should return `z` as `true` if the entry should be kept in the output of `GrB_select`, or `false` if it should not appear in the output. If the return value is not `GrB_BOOL`, it is typecasted to `GrB_BOOL` by `GrB_select`.

The type `xtype` is the GraphBLAS type of the input x of the user-defined function $z = f(x, i, j, y)$, which is used for the entry `A(i,j)` of a matrix or `v(i)` of a vector. The type may be built-in or user-defined.

The type `ytype` is the GraphBLAS type of the scalar input y of the user-defined function $z = f(x, i, j, y)$. The type may be built-in or user-defined.

6.4.2 GxB_IndexUnaryOp_new: create a named user-defined index-unary operator

```
GrB_Info GxB_IndexUnaryOp_new   // create a named user-created IndexUnaryOp
(
    GrB_IndexUnaryOp *op,           // handle for the new IndexUnary operator
    GxB_index_unary_function function,    // pointer to index_unary function
    GrB_Type ztype,                 // type of output z
    GrB_Type xtype,                 // type of input x
    GrB_Type ytype,                 // type of scalar input y
    const char *idxop_name,         // name of the user function
    const char *idxop_defn          // definition of the user function
) ;
```

Creates a named `GrB_IndexUnaryOp`. Only the first 127 characters of `idxop_name` are used. The `ixdop_defn` is a string containing the entire function itself. Currently, only the `idxop_name` is used, but future versions will rely on the `idxop_defn` when employing a JIT for better performance.

6.4.3 GrB_IndexUnaryOp_wait: wait for an index-unary operator

```
// in SuiteSparse:GraphBLAS v5.x and earlier:
GrB_Info GrB_wait               // wait for a user-defined index-unary operator
(
    GrB_IndexUnaryOp *op        // index-unary operator to wait for
) ;

// in SuiteSparse:GraphBLAS v6 (v2.0 C API):
GrB_Info GrB_wait               // wait for a user-defined binary operator
(
    GrB_IndexUnaryOp op,        // index-unary operator to wait for
    GrB_WaitMode mode           // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

After creating a user-defined select operator, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. Currently, SuiteSparse:GraphBLAS currently does nothing except to ensure that the `op` is valid.

6.4.4 GxB_IndexUnaryOp_ztype_name: return the name of the type of z

```
GrB_Info GxB_IndexUnaryOp_ztype_name    // return the type_name of x
(
    char *type_name,                    // user array of size GxB_MAX_NAME_LEN
    const GrB_IndexUnaryOp op           // index-unary operator
) ;
```

`GrB_IndexUnaryOp_ztype_name` returns the `ztype` of the index-unary operator, which is the type of z in the function $z = f(x, i, j, y)$.

6.4.5 GxB_IndexUnaryOp_xtype_name: return the name of the type of x

```
GrB_Info GxB_IndexUnaryOp_xtype_name    // return the type_name of x
(
    char *type_name,                    // user array of size GxB_MAX_NAME_LEN
    const GrB_IndexUnaryOp op           // index-unary operator
) ;
```

`GrB_IndexUnaryOp_xtype_name` returns the `xtype` of the index-unary operator, which is the type of x in the function $z = f(x, i, j, y)$. This input is used for the entry `A(i,j)` of a matrix or `v(i)` of a vector.

6.4.6 GxB_IndexUnaryOp_ytype_name: return the name of the type of scalar y

```
GrB_Info GxB_IndexUnaryOp_ytype_name    // return the type_name of the scalar y
(
    char *type_name,                    // user array of size GxB_MAX_NAME_LEN
    const GrB_IndexUnaryOp op           // index-unary operator
) ;
```

`GrB_IndexUnaryOp_ytype_name` returns the `ytype` of the index-unary operator, which is the type of the scalar y in the function $z = f(x, i, j, y)$.

6.4.7 GrB_IndexUnaryOp_free: free a user-defined index-unary operator

```
GrB_Info GrB_free               // free a user-created index-unary operator
(
    GrB_IndexUnaryOp *op        // handle of IndexUnary to free
) ;
```

`GrB_IndexUnaryOp_free` frees a user-defined index-unary operator. Either usage:

```
GrB_IndexUnaryOp_free (&op) ;
GrB_free (&op) ;
```

frees the `op` and sets `op` to `NULL`. It safely does nothing if passed a `NULL` handle, or if `op == NULL` on input. It does nothing at all if passed a built-in index-unary operator.

6.5 GraphBLAS monoids: GrB_Monoid

A *monoid* is defined on a single domain (that is, a single type), T. It consists of an associative binary operator $z = f(x, y)$ whose three operands x, y, and z are all in this same domain T (that is $T \times T \to T$). The operator must also have an identity element, or "zero" in this domain, such that $f(x, 0) = f(0, x) = x$. Recall that an associative operator $f(x, y)$ is one for which the condition $f(a, f(b, c)) = f(f(a, b), c)$ always holds. That is, operator can be applied in any order and the results remain the same. If used in a semiring, the operator must also be commutative.

Predefined binary operators that can be used to form monoids are listed in the table below. Most of these are the binary operators of predefined monoids, except that the bitwise monoids are predefined only for the unsigned integer types, not the signed integers.

Recall that T denotes any built-in type (including boolean, integer, floating point real, and complex), R denotes any non-complex type, and I denotes any integer type.

GraphBLAS operator	types (domains)	expression $z = f(x, y)$	identity	terminal
`GrB_PLUS_`T	$T \times T \to T$	$z = x + y$	0	none
`GrB_TIMES_`T	$T \times T \to T$	$z = xy$	1	0 or none (see note)
`GxB_ANY_`T	$T \times T \to T$	$z = x$ or y	any	any
`GrB_MIN_`R	$R \times R \to R$	$z = \min(x, y)$	$+\infty$	$-\infty$
`GrB_MAX_`R	$R \times R \to R$	$z = \max(x, y)$	$-\infty$	$+\infty$
`GrB_LOR`	`bool` × `bool` → `bool`	$z = x \vee y$	false	true
`GrB_LAND`	`bool` × `bool` → `bool`	$z = x \wedge y$	true	false
`GrB_LXOR`	`bool` × `bool` → `bool`	$z = x \veebar y$	false	none
`GrB_LXNOR`	`bool` × `bool` → `bool`	$z = (x == y)$	true	none
`GrB_BOR_`I	$I \times I \to I$	`z=x|y`	all bits zero	all bits one
`GrB_BAND_`I	$I \times I \to I$	`z=x&y`	all bits one	all bits zero
`GrB_BXOR_`I	$I \times I \to I$	`z=x^y`	all bits zero	none
`GrB_BXNOR_`I	$I \times I \to I$	`z=~(x^y)`	all bits one	none

The above table lists the GraphBLAS operator, its type, expression, identity value, and *terminal* value (if any). For these built-in operators, the terminal values are the *annihilators* of the function, which is the value z so that $z = f(z, y)$ regardless of the value of y. For example $\min(-\infty, y) = -\infty$ for any y. For integer domains, $+\infty$ and $-\infty$ are the largest and smallest integer in their range. With unsigned integers, the smallest value is zero, and thus `GrB_MIN_UINT8` has an identity of 255 and a terminal value of 0.

When computing with a monoid, the computation can terminate early if the terminal value arises. No further work is needed since the result will not change. This value is called the terminal value instead of the annihilator, since a user-defined operator can be created with a terminal value that is not an annihilator. See Section 6.5.3 for an example.

The `GxB_ANY_*` monoid can terminate as soon as it finds any value at all.

NOTE: The `GrB_TIMES_FP*` operators do not have a terminal value of zero, since they comply with the IEEE 754 standard, and `0*NaN` is not zero, but `NaN`. Technically, their terminal value is `NaN`, but this value is rare in practice and thus the terminal condition is

not worth checking.

The C API Specification includes 44 predefined monoids, with the naming convention `GrB_op_MONOID_type`. Forty monoids are available for the four operators `MIN`, `MAX`, `PLUS`, and `TIMES`, each with the 10 non-boolean real types. Four boolean monoids are predefined: `GrB_LOR_MONOID_BOOL`, `GrB_LAND_MONOID_BOOL`, `GrB_LXOR_MONOID_BOOL`, and `GrB_LXNOR_MONOID_BOOL`.

These all appear in SuiteSparse:GraphBLAS, which adds 33 additional predefined `GxB*` monoids, with the naming convention `GxB_op_type_MONOID`. The `ANY` operator can be used for all 13 types (including complex). The `PLUS` and `TIMES` operators are provided for both complex types, for 4 additional complex monoids. Sixteen monoids are predefined for four bitwise operators (`BOR`, `BAND`, `BXOR`, and `BNXOR`), each with four unsigned integer types (`UINT8`, `UINT16`, `UINT32`, and `UINT64`).

The next sections define the following methods for the `GrB_Monoid` object:

GraphBLAS function	purpose	Section
`GrB_Monoid_new`	create a user-defined monoid	6.5.1
`GrB_Monoid_wait`	wait for a user-defined monoid	6.5.2
`GxB_Monoid_terminal_new`	create a monoid that has a terminal value	6.5.3
`GxB_Monoid_operator`	return the monoid operator	6.5.4
`GxB_Monoid_identity`	return the monoid identity value	6.5.5
`GxB_Monoid_terminal`	return the monoid terminal value (if any)	6.5.6
`GrB_Monoid_free`	free a monoid	6.5.7

6.5.1 GrB_Monoid_new: create a monoid

```
GrB_Info GrB_Monoid_new             // create a monoid
(
    GrB_Monoid *monoid,             // handle of monoid to create
    GrB_BinaryOp op,                // binary operator of the monoid
    <type> identity                 // identity value of the monoid
) ;
```

`GrB_Monoid_new` creates a monoid. The operator, `op`, must be an associative binary operator, either built-in or user-defined.

In the definition above, `<type>` is a place-holder for the specific type of the monoid. For built-in types, it is the C type corresponding to the built-in type (see Section 6.1), such as `bool`, `int32_t`, `float`, or `double`. In this case, `identity` is a scalar value of the particular type, not a pointer. For user-defined types, `<type>` is `void *`, and thus `identity` is a not a scalar itself but a `void *` pointer to a memory location containing the identity value of the user-defined operator, `op`.

If `op` is a built-in operator with a known identity value, then the `identity` parameter is ignored, and its known identity value is used instead.

6.5.2 GrB_Monoid_wait: wait for a monoid

```
// in SuiteSparse:GraphBLAS v5.x and earlier:
GrB_Info GrB_wait               // wait for a user-defined monoid
(
    GrB_Monoid *monoid          // monoid to wait for
) ;

// in SuiteSparse:GraphBLAS v6 (v2.0 C API):
GrB_Info GrB_wait               // wait for a user-defined monoid
(
    GrB_Monoid monoid,          // monoid to wait for
    GrB_WaitMode mode           // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

After creating a user-defined monoid, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. Currently, SuiteSparse:GraphBLAS currently does nothing except to ensure that the `monoid` is valid.

6.5.3 GxB_Monoid_terminal_new: create a monoid with terminal

```
GrB_Info GxB_Monoid_terminal_new    // create a monoid that has a terminal value
(
    GrB_Monoid *monoid,             // handle of monoid to create
    GrB_BinaryOp op,                // binary operator of the monoid
    <type> identity,                // identity value of the monoid
    <type> terminal                 // terminal value of the monoid
) ;
```

`GxB_Monoid_terminal_new` is identical to `GrB_Monoid_new`, except that it allows for the specification of a *terminal value*. The `<type>` of the terminal value is the same as the `identity` parameter; see Section 6.5.1 for details.

The terminal value of a monoid is the value z for which $z = f(z, y)$ for any y, where $z = f(x, y)$ is the binary operator of the monoid. This is also called the *annihilator*, but the term *terminal value* is used here. This is because all annihilators are terminal values, but a terminal value need not be an annihilator, as described in the `MIN` example below.

If the terminal value is encountered during computation, the rest of the computations can be skipped. This can greatly improve the performance of `GrB_reduce`, and matrix multiply in specific cases (when a dot product method is used). For example, using `GrB_reduce` to compute the sum of all entries in a `GrB_FP32` matrix with e entries takes $O(e)$ time, since a monoid based on `GrB_PLUS_FP32` has no terminal value. By contrast, a reduction using `GrB_LOR` on a `GrB_BOOL` matrix can take as little as $O(1)$ time, if a `true` value is found in the matrix very early.

Monoids based on the built-in `GrB_MIN_*` and `GrB_MAX_*` operators (for any type), the boolean `GrB_LOR`, and the boolean `GrB_LAND` operators all have terminal values. For example, the identity value of `GrB_LOR` is `false`, and its terminal value is `true`. When computing a reduction of a set of boolean values to a single value, once a `true` is seen, the computation can exit early since the result is now known.

If `op` is a built-in operator with known identity and terminal values, then the `identity` and `terminal` parameters are ignored, and its known identity and terminal values are used instead.

There may be cases in which the user application needs to use a non-standard terminal value for a built-in operator. For example, suppose the matrix has type `GrB_FP32`, but all values in the matrix are known to be non-negative. The annihilator value of `MIN` is `-INFINITY`, but this will never be seen. However, the computation could terminate when finding the value zero. This is an example of using a terminal value that is not actually an annihilator, but it functions like one since the monoid will operate strictly on non-negative values.

In this case, a monoid created with `GrB_MIN_FP32` will not terminate early, because the identity and terminal inputs are ignored when using `GrB_Monoid_new` with a built-in operator as its input. To create a monoid that can terminate early, create a user-defined operator that computes the same thing as `GrB_MIN_FP32`, and then create a monoid based on this user-defined operator with a terminal value of zero and an identity of `+INFINITY`.

6.5.4 GxB_Monoid_operator: return the monoid operator

```
GrB_Info GxB_Monoid_operator        // return the monoid operator
(
    GrB_BinaryOp *op,               // returns the binary op of the monoid
    GrB_Monoid monoid               // monoid to query
) ;
```

`GxB_Monoid_operator` returns the binary operator of the monoid.

6.5.5 GxB_Monoid_identity: return the monoid identity

```
GrB_Info GxB_Monoid_identity        // return the monoid identity
(
    void *identity,                 // returns the identity of the monoid
    GrB_Monoid monoid               // monoid to query
) ;
```

`GxB_Monoid_identity` returns the identity value of the monoid. The `void *` pointer, `identity`, must be non-`NULL` and must point to a memory space of size at least equal to the size of the type of the `monoid`. The type size can be obtained via `GxB_Monoid_operator` to return the monoid additive operator, then `GxB_BinaryOp_ztype` to obtain the `ztype`, followed by `GxB_Type_size` to get its size.

6.5.6 GxB_Monoid_terminal: return the monoid terminal value

```
GrB_Info GxB_Monoid_terminal        // return the monoid terminal
(
    bool *has_terminal,             // true if the monoid has a terminal value
    void *terminal,                 // returns the terminal of the monoid
    GrB_Monoid monoid               // monoid to query
) ;
```

`GxB_Monoid_terminal` returns the terminal value of the monoid (if any). The `void *` pointer, `terminal`, must be non-`NULL` and must point to a memory space of size at least equal to the size of the type of the `monoid`. The type size can be obtained via `GxB_Monoid_operator` to return the monoid additive operator, then `GxB_BinaryOp_ztype` to obtain the `ztype`, followed by `GxB_Type_size` to get its size.

If the monoid has a terminal value, then `has_terminal` is `true`, and its value is returned in the `terminal` parameter. If it has no terminal value, then `has_terminal` is `false`, and the `terminal` parameter is not modified.

6.5.7 GrB_Monoid_free: free a monoid

```
GrB_Info GrB_free                   // free a user-created monoid
(
    GrB_Monoid *monoid              // handle of monoid to free
) ;
```

`GrB_Monoid_frees` frees a monoid. Either usage:

```
GrB_Monoid_free (&monoid) ;
GrB_free (&monoid) ;
```

frees the `monoid` and sets `monoid` to `NULL`. It safely does nothing if passed a `NULL` handle, or if `monoid == NULL` on input. It does nothing at all if passed a built-in monoid.

6.6 GraphBLAS semirings: GrB_Semiring

A *semiring* defines all the operators required to define the multiplication of two sparse matrices in GraphBLAS, $\mathbf{C} = \mathbf{AB}$. The "add" operator is a commutative and associative monoid, and the binary "multiply" operator defines a function $z = fmult(x, y)$ where the type of z matches the exactly with the monoid type. SuiteSparse:GraphBLAS includes 1,473 predefined built-in semirings. The next sections define the following methods for the `GrB_Semiring` object:

GraphBLAS function	purpose	Section
`GrB_Semiring_new`	create a user-defined semiring	6.6.1
`GrB_Semiring_wait`	wait for a user-defined semiring	6.6.2
`GxB_Semiring_add`	return the additive monoid of a semiring	6.6.3
`GxB_Semiring_multiply`	return the binary operator of a semiring	6.6.4
`GrB_Semiring_free`	free a semiring	6.6.5

6.6.1 GrB_Semiring_new: create a semiring

```
GrB_Info GrB_Semiring_new           // create a semiring
(
    GrB_Semiring *semiring,         // handle of semiring to create
    GrB_Monoid add,                 // add monoid of the semiring
    GrB_BinaryOp multiply           // multiply operator of the semiring
) ;
```

`GrB_Semiring_new` creates a new semiring, with `add` being the additive monoid and `multiply` being the binary "multiply" operator. In addition to the standard error cases, the function returns `GrB_DOMAIN_MISMATCH` if the output (`ztype`) domain of `multiply` does not match the domain of the `add` monoid.

The v2.0 C API Specification for GraphBLAS includes 124 predefined semirings, with names of the form `GrB_add_mult_SEMIRING_type`, where `add` is the operator of the additive monoid, `mult` is the multiply operator, and `type` is the type of the input x to the multiply operator, $f(x, y)$. The name of the domain for the additive monoid does not appear in the name, since it always matches the type of the output of the `mult` operator. Twelve kinds of `GrB*` semirings are available for all 10 real, non-boolean types: `PLUS_TIMES`, `PLUS_MIN`, `MIN_PLUS`, `MIN_TIMES`, `MIN_FIRST`, `MIN_SECOND`, `MIN_MAX`, `MAX_PLUS`, `MAX_TIMES`, `MAX_FIRST`, `MAX_SECOND`, and `MAX_MIN`. Four semirings are for boolean types only: `LOR_LAND`, `LAND_LOR`, `LXOR_LAND`, and `LXNOR_LOR`.

SuiteSparse:GraphBLAS pre-defines 1,553 semirings from built-in types and operators, listed below. The naming convention is `GxB_add_mult_type`. The 124 `GrB*` semirings are a subset of the list below, included with two names: `GrB*` and `GxB*`. If the `GrB*` name is provided, its use is preferred, for portability to other GraphBLAS implementations.

- 1000 semirings with a multiplier $T \times T \to T$ where T is any of the 10 non-Boolean, real types, from the complete cross product of:
  - 5 monoids (`MIN`, `MAX`, `PLUS`, `TIMES`, `ANY`)

  - 20 multiply operators (`FIRST`, `SECOND`, `PAIR` (same as `ONEB`), `MIN`, `MAX`, `PLUS`, `MINUS`, `RMINUS`, `TIMES`, `DIV`, `RDIV`, `ISEQ`, `ISNE`, `ISGT`, `ISLT`, `ISGE`, `ISLE`, `LOR`, `LAND`, `LXOR`).
  - 10 non-Boolean types, T

- 300 semirings with a comparator $T \times T \to$ `bool`, where T is non-Boolean and real, from the complete cross product of:
  - 5 Boolean monoids (`LAND`, `LOR`, `LXOR`, `EQ`, `ANY`)
  - 6 multiply operators (`EQ`, `NE`, `GT`, `LT`, `GE`, `LE`)
  - 10 non-Boolean types, T

- 55 semirings with purely Boolean types, `bool` $\times$ `bool` $\to$ `bool`, from the complete cross product of:
  - 5 Boolean monoids (`LAND`, `LOR`, `LXOR`, `EQ`, `ANY`)
  - 11 multiply operators (`FIRST`, `SECOND`, `PAIR` (same as `ONEB`), `LOR`, `LAND`, `LXOR`, `EQ`, `GT`, `LT`, `GE`, `LE`)

- 54 complex semirings, $Z \times Z \to Z$ where Z is `GxB_FC32` (single precision complex) or `GxB_FC64` (double precision complex):
  - 3 complex monoids (`PLUS`, `TIMES`, `ANY`)
  - 9 complex multiply operators (`FIRST`, `SECOND`, `PAIR` (same as `ONEB`), `PLUS`, `MINUS`, `TIMES`, `DIV`, `RDIV`, `RMINUS`)
  - 2 complex types, Z

- 64 bitwise semirings, $U \times U \to U$ where U is an unsigned integer.
  - 4 bitwise monoids (`BOR`, `BAND`, `BXOR`, `BXNOR`)
  - 4 bitwise multiply operators (the same list)
  - 4 unsigned integer types

- 80 positional semirings, $X \times X \to T$ where T is `INT32` or `INT64`:
  - 5 monoids (`MIN`, `MAX`, `PLUS`, `TIMES`, `ANY`)
  - 8 positional operators (`FIRSTI`, `FIRSTI1`, `FIRSTJ`, `FIRSTJ1`, `SECONDI`, `SECONDI1`, `SECONDJ`, `SECONDJ1`)
  - 2 integer types (`INT32`, `INT64`)

6.6.2 GrB_Semiring_wait: wait for a semiring

```
// in SuiteSparse:GraphBLAS v5.x and earlier:
GrB_Info GrB_wait               // wait for a user-defined semiring
(
    GrB_Semiring *semiring      // semiring to wait for
) ;

// in SuiteSparse:GraphBLAS v6 (v2.0 C API):
GrB_Info GrB_wait               // wait for a user-defined semiring
(
    GrB_Semiring semiring,      // semiring to wait for
    GrB_WaitMode mode           // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

After creating a user-defined semiring, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. Currently, SuiteSparse:GraphBLAS currently does nothing except to ensure that the `semiring` is valid.

6.6.3 GxB_Semiring_add: return the additive monoid of a semiring

```
GrB_Info GxB_Semiring_add           // return the add monoid of a semiring
(
    GrB_Monoid *add,                // returns add monoid of the semiring
    GrB_Semiring semiring           // semiring to query
) ;
```

`GxB_Semiring_add` returns the additive monoid of a semiring.

6.6.4 GxB_Semiring_multiply: return multiply operator of a semiring

```
GrB_Info GxB_Semiring_multiply      // return multiply operator of a semiring
(
    GrB_BinaryOp *multiply,         // returns multiply operator of the semiring
    GrB_Semiring semiring           // semiring to query
) ;
```

`GxB_Semiring_multiply` returns the binary multiplicative operator of a semiring.

6.6.5 GrB_Semiring_free: free a semiring

```
GrB_Info GrB_free                   // free a user-created semiring
(
    GrB_Semiring *semiring          // handle of semiring to free
) ;
```

`GrB_Semiring_free` frees a semiring. Either usage:

```
GrB_Semiring_free (&semiring) ;
GrB_free (&semiring) ;
```

frees the `semiring` and sets `semiring` to `NULL`. It safely does nothing if passed a `NULL` handle, or if `semiring == NULL` on input. It does nothing at all if passed a built-in semiring.

6.7 GraphBLAS scalars: GrB_Scalar

This section describes a set of methods that create, modify, query, and destroy a GraphBLAS scalar, `GrB_Scalar`:

GraphBLAS function	purpose	Section
`GrB_Scalar_new`	create a scalar	6.7.1
`GrB_Scalar_wait`	wait for a scalar	6.7.2
`GrB_Scalar_dup`	copy a scalar	6.7.3
`GrB_Scalar_clear`	clear a scalar of its entry	6.7.4
`GrB_Scalar_nvals`	return number of entries in a scalar	6.7.5
`GxB_Scalar_type_name`	return name of the type of a scalar	6.7.6
`GrB_Scalar_setElement`	set the single entry of a scalar	6.7.7
`GrB_Scalar_extractElement`	get the single entry from a scalar	6.7.8
`GxB_Scalar_memoryUsage`	memory used by a scalar	6.7.9
`GrB_Scalar_free`	free a scalar	6.7.10

6.7.1 GrB_Scalar_new: create a scalar

```
GrB_Info GrB_Scalar_new     // create a new GrB_Scalar with no entry
(
    GrB_Scalar *s,          // handle of GrB_Scalar to create
    GrB_Type type           // type of GrB_Scalar to create
) ;
```

`GrB_Scalar_new` creates a new scalar with no entry in it, of the given type. This is analogous to Octave/MATLAB statement `s = sparse(0)`, except that GraphBLAS can create scalars any type. The pattern of the new scalar is empty.

6.7.2 GrB_Scalar_wait: wait for a scalar

```
// in SuiteSparse:GraphBLAS v5.x and earlier:
GrB_Info GrB_wait               // wait for a scalar
(
    GrB_Scalar *s               // scalar to wait for
) ;

// in SuiteSparse:GraphBLAS v6 (v2.0 C API):
GrB_Info GrB_wait               // wait for a scalar
(
    GrB_Scalar s,               // scalar to wait for
    GrB_WaitMode mode           // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

In non-blocking mode, the computations for a `GrB_Scalar` may be delayed. In this case, the scalar is not yet safe to use by multiple independent user threads. A user application may force completion of a scalar `s` via `GrB_Scalar_wait(&s)` (in v5.2.0), or `GrB_Scalar_wait(s,mode)` (in v6.0.0). With a `mode` of `GrB_MATERIALIZE`, all pending computations are finished, and different user threads may simultaneously call GraphBLAS operations that use the scalar `s` as an input parameter.

6.7.3 GrB_Scalar_dup: copy a scalar

```
GrB_Info GrB_Scalar_dup     // make an exact copy of a GrB_Scalar
(
    GrB_Scalar *s,          // handle of output GrB_Scalar to create
    const GrB_Scalar t      // input GrB_Scalar to copy
) ;
```

`GrB_Scalar_dup` makes a deep copy of a scalar. In GraphBLAS, it is possible, and valid, to write the following:

```
GrB_Scalar t, s ;
GrB_Scalar_new (&t, GrB_FP64) ;
s = t ;                         // s is a shallow copy of t
```

Then `s` and `t` can be used interchangeably. However, only a pointer reference is made, and modifying one of them modifies both, and freeing one of them leaves the other as a dangling handle that should not be used. If two different scalars are needed, then this should be used instead:

```
GrB_Scalar t, s ;
GrB_Scalar_new (&t, GrB_FP64) ;
GrB_Scalar_dup (&s, t) ;        // like s = t, but making a deep copy
```

Then `s` and `t` are two different scalars that currently have the same value, but they do not depend on each other. Modifying one has no effect on the other.

6.7.4 GrB_Scalar_clear: clear a scalar of its entry

```
GrB_Info GrB_Scalar_clear   // clear a GrB_Scalar of its entry
(                           // type remains unchanged.
    GrB_Scalar s            // GrB_Scalar to clear
) ;
```

`GrB_Scalar_clear` clears the entry from a scalar. The pattern of `s` is empty, just as if it were created fresh with `GrB_Scalar_new`. Analogous with `s = sparse (0)` in Octave/MATLAB. The type of `s` does not change. Any pending updates to the scalar are discarded.

6.7.5 GrB_Scalar_nvals: return the number of entries in a scalar

```
GrB_Info GrB_Scalar_nvals   // get the number of entries in a GrB_Scalar
(
    GrB_Index *nvals,       // GrB_Scalar has nvals entries (0 or 1)
    const GrB_Scalar s      // GrB_Scalar to query
) ;
```

`GrB_Scalar_nvals` returns the number of entries in a scalar, which is either 0 or 1. Roughly analogous to `nvals = nnz(s)` in Octave/MATLAB, except that the implicit value in GraphBLAS need not be zero and `nnz` (short for "number of nonzeros") in MATLAB is better described as "number of entries" in GraphBLAS.

6.7.6 GxB_Scalar_type_name: return name of the type of a scalar

```
GrB_Info GxB_Scalar_type_name      // return the name of the type of a scalar
(
    char *type_name,        // name of the type (char array of size at least
                            // GxB_MAX_NAME_LEN, owned by the user application).
    const GrB_Scalar s      // GrB_Scalar to query
) ;
```

`GxB_Scalar_type_name` returns the name of the type of a scalar. Analogous to `type = class (s)` in MATLAB.

6.7.7 GrB_Scalar_setElement: set the single entry of a scalar

```
GrB_Info GrB_Scalar_setElement          // s = x
(
    GrB_Scalar s,                       // GrB_Scalar to modify
    <type> x                            // user scalar to assign to s
) ;
```

`GrB_Scalar_setElement` sets the single entry in a scalar, like `s = sparse(x)` in MATLAB notation. For further details of this function, see `GrB_Matrix_setElement` in Section 6.9.11. If an error occurs, `GrB_error(&err,s)` returns details about the error. The scalar `x` can be any non-opaque C scalar corresponding to a built-in type, or `void *` for a user-defined type. It cannot be a `GrB_Scalar`.

6.7.8 GrB_Scalar_extractElement: get the single entry from a scalar

```
GrB_Info GrB_Scalar_extractElement  // x = s
(
    <type> *x,                      // user scalar extracted
    const GrB_Scalar s              // GrB_Sclar to extract an entry from
) ;
```

`GrB_Scalar_extractElement` extracts the single entry from a sparse scalar, like `x = full(s)` in MATLAB. Further details of this method are discussed in Section 6.9.12, which discusses `GrB_Matrix_extractElement`. **NOTE:** if no entry is present in the scalar `s`, then `x` is not modified, and the return value of `GrB_Scalar_extractElement` is `GrB_NO_VALUE`.

6.7.9 GxB_Scalar_memoryUsage: memory used by a scalar

```
GrB_Info GxB_Scalar_memoryUsage  // return # of bytes used for a scalar
(
    size_t *size,           // # of bytes used by the scalar s
    const GrB_Scalar s      // GrB_Scalar to query
```

```
) ;
```

Returns the memory space required for a scalar, in bytes.

6.7.10 GrB_Scalar_free: free a scalar

```
GrB_Info GrB_free           // free a GrB_Scalar
(
    GrB_Scalar *s           // handle of GrB_Scalar to free
) ;
```

`GrB_Scalar_free` frees a scalar. Either usage:

```
GrB_Scalar_free (&s) ;
GrB_free (&s) ;
```

frees the scalar `s` and sets `s` to `NULL`. It safely does nothing if passed a `NULL` handle, or if `s == NULL` on input. Any pending updates to the scalar are abandoned.

6.8 GraphBLAS vectors: GrB_Vector

This section describes a set of methods that create, modify, query, and destroy a GraphBLAS sparse vector, `GrB_Vector`:

GraphBLAS function	purpose	Section
`GrB_Vector_new`	create a vector	6.8.1
`GrB_Vector_wait`	wait for a vector	6.8.2
`GrB_Vector_dup`	copy a vector	6.8.3
`GrB_Vector_clear`	clear a vector of all entries	6.8.4
`GrB_Vector_size`	size of a vector	6.8.5
`GrB_Vector_nvals`	number of entries in a vector	6.8.6
`GxB_Vector_type_name`	name of the type of a vector	6.8.7
`GrB_Vector_build`	build a vector from tuples	6.8.8
`GxB_Vector_build_Scalar`	build a vector from tuples	6.8.9
`GrB_Vector_setElement`	add an entry to a vector	6.8.10
`GrB_Vector_extractElement`	get an entry from a vector	6.8.11
`GrB_Vector_removeElement`	remove an entry from a vector	6.8.12
`GrB_Vector_extractTuples`	get all entries from a vector	6.8.13
`GrB_Vector_resize`	resize a vector	6.8.14
`GxB_Vector_diag`	extract a diagonal from a matrix	6.8.15
`GxB_Vector_iso`	query iso status	6.8.16
`GxB_Vector_memoryUsage`	memory used by a vector	6.8.17
`GrB_Vector_free`	free a vector	6.8.18
`GxB_Vector_serialize`	serialize a vector	6.10.1
`GxB_Vector_deserialize`	deserialize a vector	6.10.2
`GxB_Vector_pack_CSC`	pack in CSC format	6.11.1
`GxB_Vector_unpack_CSC`	unpack in CSC format	6.11.2
`GxB_Vector_pack_Bitmap`	pack in bitmap format	6.11.3
`GxB_Vector_unpack_Bitmap`	unpack in bitmap format	6.11.4
`GxB_Vector_pack_Full`	pack in full format	6.11.5
`GxB_Vector_unpack_Full`	unpack in full format	6.11.6
`GxB_Vector_sort`	sort a vector	6.13.1

Refer to Section 6.10 for serialization/deserialization methods, Section 6.11 for pack/unpack methods, and to Section 6.13 for sorting methods.

6.8.1 GrB_Vector_new: create a vector

```
GrB_Info GrB_Vector_new     // create a new vector with no entries
(
    GrB_Vector *v,          // handle of vector to create
    GrB_Type type,          // type of vector to create
    GrB_Index n             // vector dimension is n-by-1
) ;
```

`GrB_Vector_new` creates a new `n`-by-`1` sparse vector with no entries in it, of the given type. This is analogous to Octave/MATLAB statement `v = sparse (n,1)`, except that GraphBLAS can create sparse vectors any type. The pattern of the new vector is empty.

SPEC: `n` may be zero, as an extension to the specification.

6.8.2 GrB_Vector_wait: wait for a vector

```
// in SuiteSparse:GraphBLAS v5.x and earlier:
GrB_Info GrB_wait               // wait for a vector
(
    GrB_Vector *w               // vector to wait for
) ;

// in SuiteSparse:GraphBLAS v6 (v2.0 C API):
GrB_Info GrB_wait               // wait for a vector
(
    GrB_Vector w,               // vector to wait for
    GrB_WaitMode mode           // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

In non-blocking mode, the computations for a `GrB_Vector` may be delayed. In this case, the vector is not yet safe to use by multiple independent user threads. A user application may force completion of a vector `w` via `GrB_Vector_wait(&w)` (in v5.2.0), or `GrB_Vector_wait(w,mode)` (in v6.0.0). With a `mode` of `GrB_MATERIALIZE`, all pending computations are finished, and different user threads may simultaneously call GraphBLAS operations that use the vector `w` as an input parameter.

6.8.3 GrB_Vector_dup: copy a vector

```
GrB_Info GrB_Vector_dup     // make an exact copy of a vector
(
    GrB_Vector *w,          // handle of output vector to create
    const GrB_Vector u      // input vector to copy
) ;
```

`GrB_Vector_dup` makes a deep copy of a sparse vector. In GraphBLAS, it is possible, and valid, to write the following:

```
GrB_Vector u, w ;
GrB_Vector_new (&u, GrB_FP64, n) ;
w = u ;                         // w is a shallow copy of u
```

Then `w` and `u` can be used interchangeably. However, only a pointer reference is made, and modifying one of them modifies both, and freeing one of them leaves the other as a dangling handle that should not be used. If two different vectors are needed, then this should be used instead:

```
GrB_Vector u, w ;
GrB_Vector_new (&u, GrB_FP64, n) ;
GrB_Vector_dup (&w, u) ;        // like w = u, but making a deep copy
```

Then `w` and `u` are two different vectors that currently have the same set of values, but they do not depend on each other. Modifying one has no effect on the other.

6.8.4 GrB_Vector_clear: clear a vector of all entries

```
GrB_Info GrB_Vector_clear   // clear a vector of all entries;
(                           // type and dimension remain unchanged.
    GrB_Vector v            // vector to clear
) ;
```

`GrB_Vector_clear` clears all entries from a vector. All values `v(i)` are now equal to the implicit value, depending on what semiring ring is used to perform computations on the vector. The pattern of `v` is empty, just as if it were created fresh with `GrB_Vector_new`. Analogous with `v (:) = sparse(0)` in MATLAB. The type and dimension of `v` do not change. Any pending updates to the vector are discarded.

6.8.5 GrB_Vector_size: return the size of a vector

```
GrB_Info GrB_Vector_size    // get the dimension of a vector
(
    GrB_Index *n,           // vector dimension is n-by-1
```

```
    const GrB_Vector v      // vector to query
) ;
```

GrB_Vector_size returns the size of a vector (the number of rows). Analogous to `n = length(v)` or `n = size(v,1)` in MATLAB.

6.8.6 GrB_Vector_nvals: return the number of entries in a vector

```
GrB_Info GrB_Vector_nvals   // get the number of entries in a vector
(
    GrB_Index *nvals,       // vector has nvals entries
    const GrB_Vector v      // vector to query
) ;
```

GrB_Vector_nvals returns the number of entries in a vector. Roughly analogous to `nvals = nnz(v)` in MATLAB, except that the implicit value in GraphBLAS need not be zero and `nnz` (short for "number of nonzeros") in MATLAB is better described as "number of entries" in GraphBLAS.

6.8.7 GxB_Vector_type_name: return name of the type of a vector

```
GrB_Info GxB_Vector_type_name      // return the name of the type of a vector
(
    char *type_name,        // name of the type (char array of size at least
                            // GxB_MAX_NAME_LEN, owned by the user application).
    const GrB_Vector v      // vector to query
) ;
```

GxB_Vector_type_name returns the name of the type of a vector. Analogous to `type = class (v)` in MATLAB.

6.8.8 GrB_Vector_build: build a vector from a set of tuples

```
GrB_Info GrB_Vector_build           // build a vector from (I,X) tuples
(
    GrB_Vector w,                   // vector to build
    const GrB_Index *I,             // array of row indices of tuples
    const <type> *X,                // array of values of tuples
    GrB_Index nvals,                // number of tuples
    const GrB_BinaryOp dup          // binary function to assemble duplicates
) ;
```

`GrB_Vector_build` constructs a sparse vector `w` from a set of tuples, `I` and `X`, each of length `nvals`. The vector `w` must have already been initialized with `GrB_Vector_new`, and it must have no entries in it before calling `GrB_Vector_build`. This function is just like `GrB_Matrix_build` (see Section 6.9.9), except that it builds a sparse vector instead of a sparse matrix. For a description of what `GrB_Vector_build` does, refer to `GrB_Matrix_build`. For a vector, the list of column indices `J` in `GrB_Matrix_build` is implicitly a vector of length `nvals` all equal to zero. Otherwise the methods are identical.

If `dup` is `NULL`, any duplicates result in an error. If `dup` is the special binary operator `GxB_IGNORE_DUP`, then any duplicates are ignored. If duplicates appear, the last one in the list of tuples is taken and the prior ones ignored. This is not an error.

SPEC: Results are defined even if `dup` is non-associative.

6.8.9 GxB_Vector_build_Scalar: build a vector from a set of tuples

```
GrB_Info GxB_Vector_build_Scalar    // build a vector from (i,scalar) tuples
(
    GrB_Vector w,                   // vector to build
    const GrB_Index *I,             // array of row indices of tuples
    GrB_Scalar scalar,              // value for all tuples
    GrB_Index nvals                 // number of tuples
) ;
```

`GxB_Vector_build_Scalar` constructs a sparse vector `w` from a set of tuples defined by the index array `I` of length `nvals`, and a scalar. The scalar is the value of all of the tuples. Unlike `GrB_Vector_build`, there is no `dup` operator to handle duplicate entries. Instead, any duplicates are silently ignored (if the number of duplicates is desired, simply compare the input `nvals` with the value returned by `GrB_Vector_nvals` after the vector is constructed). All entries in the sparsity pattern of `w` are identical, and equal to the input scalar value.

6.8.10 GrB_Vector_setElement: add an entry to a vector

```
GrB_Info GrB_Vector_setElement          // w(i) = x
(
    GrB_Vector w,                       // vector to modify
    <type> x,                           // scalar to assign to w(i)
    GrB_Index i                         // index
) ;
```

`GrB_Vector_setElement` sets a single entry in a vector, `w(i) = x`. The operation is exactly like setting a single entry in an `n`-by-1 matrix, `A(i,0) = x`, where the column index for a vector is implicitly `j=0`. For further details of this function, see `GrB_Matrix_setElement` in Section 6.9.11. If an error occurs, `GrB_error(&err,w)` returns details about the error.

6.8.11 GrB_Vector_extractElement: get an entry from a vector

```
GrB_Info GrB_Vector_extractElement  // x = v(i)
(
    <type> *x,                  // scalar extracted (non-opaque, C scalar)
    const GrB_Vector v,         // vector to extract an entry from
    GrB_Index i                 // index
) ;

GrB_Info GrB_Vector_extractElement  // x = v(i)
(
    GrB_Scalar x,               // GrB_Scalar extracted
    const GrB_Vector v,         // vector to extract an entry from
    GrB_Index i                 // index
) ;
```

`GrB_Vector_extractElement` extracts a single entry from a vector, `x = v(i)`. The method is identical to extracting a single entry `x = A(i,0)` from an `n`-by-1 matrix; see Section 6.9.12.

6.8.12 GrB_Vector_removeElement: remove an entry from a vector

```
GrB_Info GrB_Vector_removeElement
(
    GrB_Vector w,                   // vector to remove an entry from
    GrB_Index i                     // index
) ;
```

`GrB_Vector_removeElement` removes a single entry `w(i)` from a vector. If no entry is present at `w(i)`, then the vector is not modified. If an error occurs, `GrB_error(&err,w)` returns details about the error.

6.8.13 GrB_Vector_extractTuples: get all entries from a vector

```
GrB_Info GrB_Vector_extractTuples           // [I,~,X] = find (v)
(
    GrB_Index *I,               // array for returning row indices of tuples
    <type> *X,                  // array for returning values of tuples
    GrB_Index *nvals,           // I, X size on input; # tuples on output
    const GrB_Vector v          // vector to extract tuples from
) ;
```

`GrB_Vector_extractTuples` extracts all tuples from a sparse vector, analogous to `[I,~,X] = find(v)` in Octave/MATLAB. This function is identical to its `GrB_Matrix_extractTuples` counterpart, except that the array of column indices `J` does not appear in this function. Refer to Section 6.9.14 where further details of this function are described.

6.8.14 GrB_Vector_resize: resize a vector

```
GrB_Info GrB_Vector_resize      // change the size of a vector
(
    GrB_Vector u,               // vector to modify
    GrB_Index nrows_new         // new number of rows in vector
) ;
```

`GrB_Vector_resize` changes the size of a vector. If the dimension decreases, entries that fall outside the resized vector are deleted.

6.8.15 GxB_Vector_diag: extract a diagonal from a matrix

```
GrB_Info GxB_Vector_diag    // extract a diagonal from a matrix
(
    GrB_Vector v,                   // output vector
    const GrB_Matrix A,             // input matrix
    int64_t k,
    const GrB_Descriptor desc       // unused, except threading control
) ;
```

`GxB_Vector_diag` extracts a vector `v` from an input matrix `A`, which may be rectangular. If `k` = 0, the main diagonal of `A` is extracted; `k` > 0 denotes diagonals above the main diagonal of `A`, and `k` < 0 denotes diagonals below the main diagonal of `A`. Let `A` have dimension m-by-n. If `k` is in the range 0 to $n-1$, then `v` has length $\min(m, n-k)$. If `k` is negative and in the range -1 to $-m+1$, then `v` has length $\min(m+k, n)$. If `k` is outside these ranges, `v` has length 0 (this is not an error). This function computes the same thing as the Octave/MATLAB statement `v=diag(A,k)` when `A` is a matrix, except that `GxB_Vector_diag` can also do typecasting.

The vector `v` must already exist on input, and `GrB_Vector_size (&len,v)` must return `len` = 0 if `k` $\geq n$ or `k` $\leq -m$, `len` = $\min(m, n-k)$ if `k` is in the range 0 to $n-1$, and `len` = $\min(m+k, n)$ if `k` is in the range -1 to $-m+1$. Any existing entries in `v` are discarded. The type of `v` is preserved, so that if the type of `A` and `v` differ, the entries are typecasted into the type of `v`. Any settings made to `v` by `GxB_Vector_Option_set` (bitmap switch and sparsity control) are unchanged.

6.8.16 GxB_Vector_iso: query iso status of a vector

```
GrB_Info GxB_Vector_iso     // return iso status of a vector
(
    bool *iso,              // true if the vector is iso-valued
    const GrB_Vector v      // vector to query
) ;
```

Returns the true if the vector is iso-valued, false otherwise.

6.8.17 GxB_Vector_memoryUsage: memory used by a vector

```
GrB_Info GxB_Vector_memoryUsage  // return # of bytes used for a vector
(
    size_t *size,           // # of bytes used by the vector v
    const GrB_Vector v      // vector to query
) ;
```

Returns the memory space required for a vector, in bytes.

6.8.18 GrB_Vector_free: free a vector

```
GrB_Info GrB_free           // free a vector
(
    GrB_Vector *v           // handle of vector to free
) ;
```

`GrB_Vector_free` frees a vector. Either usage:

```
GrB_Vector_free (&v) ;
GrB_free (&v) ;
```

frees the vector `v` and sets `v` to `NULL`. It safely does nothing if passed a `NULL` handle, or if `v == NULL` on input. Any pending updates to the vector are abandoned.

6.9 GraphBLAS matrices: GrB_Matrix

This section describes a set of methods that create, modify, query, and destroy a GraphBLAS sparse matrix, `GrB_Matrix`:

GraphBLAS function	purpose	Section
`GrB_Matrix_new`	create a matrix	6.9.1
`GrB_Matrix_wait`	wait for a matrix	6.9.2
`GrB_Matrix_dup`	copy a matrix	6.9.3
`GrB_Matrix_clear`	clear a matrix of all entries	6.9.4
`GrB_Matrix_nrows`	number of rows of a matrix	6.9.5
`GrB_Matrix_ncols`	number of columns of a matrix	6.9.6
`GrB_Matrix_nvals`	number of entries in a matrix	6.9.7
`GxB_Matrix_type_name`	type of a matrix	6.9.8
`GrB_Matrix_build`	build a matrix from tuples	6.9.9
`GxB_Matrix_build_Scalar`	build a matrix from tuples	6.9.10
`GrB_Matrix_setElement`	add an entry to a matrix	6.9.11
`GrB_Matrix_extractElement`	get an entry from a matrix	6.9.12
`GrB_Matrix_removeElement`	remove an entry from a matrix	6.9.13
`GrB_Matrix_extractTuples`	get all entries from a matrix	6.9.14
`GrB_Matrix_resize`	resize a matrix	6.9.15
`GxB_Matrix_concat`	concatenate matrices	6.9.16
`GxB_Matrix_split`	split a matrix into matrices	6.9.17
`GrB_Matrix_diag`	diagonal matrix from vector	6.9.18
`GxB_Matrix_diag`	diagonal matrix from vector	6.9.19
`GxB_Matrix_iso`	query iso status	6.9.20
`GxB_Matrix_memoryUsage`	memory used by a matrix	6.9.21
`GrB_Matrix_free`	free a matrix	6.9.22
`GrB_Matrix_serializeSize`	return size of serialized matrix	6.10.3
`GrB_Matrix_serialize`	serialize a matrix	6.10.4
`GxB_Matrix_serialize`	serialize a matrix	6.10.5
`GrB_Matrix_deserialize`	deserialize a matrix	6.10.6
`GxB_Matrix_deserialize`	deserialize a matrix	6.10.7

GraphBLAS function	purpose	Section
GxB_Matrix_pack_CSR	pack CSR	6.11.7
GxB_Matrix_unpack_CSR	unpack CSR	6.11.8
GxB_Matrix_pack_CSC	pack CSC	6.11.9
GxB_Matrix_unpack_CSC	unpack CSC	6.11.10
GxB_Matrix_pack_HyperCSR	pack HyperCSR	6.11.11
GxB_Matrix_unpack_HyperCSR	unpack HyperCSR	6.11.12
GxB_Matrix_pack_HyperCSC	pack HyperCSC	6.11.13
GxB_Matrix_unpack_HyperCSC	unpack HyperCSC	6.11.14
GxB_Matrix_pack_BitmapR	pack BitmapR	6.11.15
GxB_Matrix_unpack_BitmapR	unpack BitmapR	6.11.16
GxB_Matrix_pack_BitmapC	pack BitmapC	6.11.17
GxB_Matrix_unpack_BitmapC	unpack BitmapC	6.11.18
GxB_Matrix_pack_FullR	pack FullR	6.11.19
GxB_Matrix_unpack_FullR	unpack FullR	6.11.20
GxB_Matrix_pack_FullC	pack FullC	6.11.21
GxB_Matrix_unpack_FullC	unpack FullC	6.11.22
GrB_Matrix_import	import in various formats	6.12.1
GrB_Matrix_export	export in various formats	6.12.2
GrB_Matrix_exportSize	array sizes for export	6.12.3
GrB_Matrix_exportHint	hint best export format	6.12.4
GxB_Matrix_sort	sort a matrix	6.13.2

Refer to Section 6.10 for serialization/deserialization methods, Section 6.11 for GxBpack/unpack methods, Section 6.12 for GrB import/export methods, and Section 6.13 for sorting methods.

6.9.1 GrB_Matrix_new: create a matrix

```
GrB_Info GrB_Matrix_new     // create a new matrix with no entries
(
    GrB_Matrix *A,          // handle of matrix to create
    GrB_Type type,          // type of matrix to create
    GrB_Index nrows,        // matrix dimension is nrows-by-ncols
    GrB_Index ncols
) ;
```

GrB_Matrix_new creates a new nrows-by-ncols sparse matrix with no entries in it, of the given type. This is analogous to the MATLAB statement A = sparse (nrows, ncols), except that GraphBLAS can create sparse matrices of any type.

By default, matrices of size nrows-by-1 are held by column, regardless of the global setting controlled by GxB_set (GxB_FORMAT, ...), for any value of nrows. Matrices of size 1-by-ncols with ncols not equal to 1 are held by row, regardless of this global setting. The global setting only affects matrices with both m > 1 and n > 1. Empty matrices (0-by-0) are also controlled by the global setting.

Once a matrix is created, its format (by-row or by-column) can be arbitrarily changed with `GxB_set (A, GxB_FORMAT, fmt)` with `fmt` equal to `GxB_BY_COL` or `GxB_BY_ROW`.

SPEC: `nrows` and/or `ncols` may be zero, as an extension to the specification.

6.9.2 GrB_Matrix_wait: wait for a matrix

```
// in SuiteSparse:GraphBLAS v5.x and earlier:
GrB_Info GrB_wait               // wait for a matrix
(
    GrB_Matrix *C               // matrix to wait for
) ;

// in SuiteSparse:GraphBLAS v6 (v2.0 C API):
GrB_Info GrB_wait               // wait for a matrix
(
    GrB_Matrix C,               // matrix to wait for
    GrB_WaitMode mode           // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

In non-blocking mode, the computations for a `GrB_Matrix` may be delayed. In this case, the matrix is not yet safe to use by multiple independent user threads. A user application may force completion of a matrix `C` via `GrB_Matrix_wait(&C)` (in v5.2.0), or `GrB_Matrix_wait(C,mode)` (in v6.0.0). With a `mode` of `GrB_MATERIALIZE`, all pending computations are finished, and different user threads may simultaneously call GraphBLAS operations that use the matrix `C` as an input parameter.

6.9.3 GrB_Matrix_dup: copy a matrix

```
GrB_Info GrB_Matrix_dup     // make an exact copy of a matrix
(
    GrB_Matrix *C,          // handle of output matrix to create
    const GrB_Matrix A      // input matrix to copy
) ;
```

`GrB_Matrix_dup` makes a deep copy of a sparse matrix. In GraphBLAS, it is possible, and valid, to write the following:

```
GrB_Matrix A, C ;
GrB_Matrix_new (&A, GrB_FP64, n) ;
C = A ;                         // C is a shallow copy of A
```

Then `C` and `A` can be used interchangeably. However, only a pointer reference is made, and modifying one of them modifies both, and freeing one of them leaves the other as a dangling handle that should not be used. If two different matrices are needed, then this should be used instead:

```
GrB_Matrix A, C ;
GrB_Matrix_new (&A, GrB_FP64, n) ;
GrB_Matrix_dup (&C, A) ;        // like C = A, but making a deep copy
```

Then `C` and `A` are two different matrices that currently have the same set of values, but they do not depend on each other. Modifying one has no effect on the other.

6.9.4 GrB_Matrix_clear: clear a matrix of all entries

```
GrB_Info GrB_Matrix_clear   // clear a matrix of all entries;
(                           // type and dimensions remain unchanged
    GrB_Matrix A            // matrix to clear
) ;
```

`GrB_Matrix_clear` clears all entries from a matrix. All values `A(i,j)` are now equal to the implicit value, depending on what semiring ring is used to perform computations on the matrix. The pattern of `A` is empty, just as if it were created fresh with `GrB_Matrix_new`. Analogous with `A (:,:) = 0` in MATLAB. The type and dimensions of `A` do not change. Any pending updates to the matrix are discarded.

6.9.5 GrB_Matrix_nrows: return the number of rows of a matrix

```
GrB_Info GrB_Matrix_nrows   // get the number of rows of a matrix
(
    GrB_Index *nrows,       // matrix has nrows rows
```

```
    const GrB_Matrix A      // matrix to query
) ;
```

`GrB_Matrix_nrows` returns the number of rows of a matrix (`nrows=size(A,1)` in MATLAB).

6.9.6 GrB_Matrix_ncols: return the number of columns of a matrix

```
GrB_Info GrB_Matrix_ncols   // get the number of columns of a matrix
(
    GrB_Index *ncols,       // matrix has ncols columns
    const GrB_Matrix A      // matrix to query
) ;
```

`GrB_Matrix_ncols` returns the number of columns of a matrix (`ncols=size(A,2)` in MATLAB).

6.9.7 GrB_Matrix_nvals: return the number of entries in a matrix

```
GrB_Info GrB_Matrix_nvals   // get the number of entries in a matrix
(
    GrB_Index *nvals,       // matrix has nvals entries
    const GrB_Matrix A      // matrix to query
) ;
```

`GrB_Matrix_nvals` returns the number of entries in a matrix. Roughly analogous to `nvals = nnz(A)` in MATLAB, except that the implicit value in GraphBLAS need not be zero and `nnz` (short for "number of nonzeros") in MATLAB is better described as "number of entries" in GraphBLAS.

6.9.8 GxB_Matrix_type_name: return name of the type of a matrix

```
GrB_Info GxB_Matrix_type_name      // return the name of the type of a matrix
(
    char *type_name,        // name of the type (char array of size at least
                            // GxB_MAX_NAME_LEN, owned by the user application).
    const GrB_Matrix A      // matrix to query
) ;
```

`GxB_Matrix_type_name` returns the name of the type of a matrix, like `type=class(A)` in MATLAB.

6.9.9 GrB_Matrix_build: build a matrix from a set of tuples

```
GrB_Info GrB_Matrix_build           // build a matrix from (I,J,X) tuples
(
    GrB_Matrix C,                   // matrix to build
    const GrB_Index *I,             // array of row indices of tuples
    const GrB_Index *J,             // array of column indices of tuples
    const <type> *X,                // array of values of tuples
    GrB_Index nvals,                // number of tuples
    const GrB_BinaryOp dup          // binary function to assemble duplicates
) ;
```

`GrB_Matrix_build` constructs a sparse matrix `C` from a set of tuples, `I`, `J`, and `X`, each of length `nvals`. The matrix `C` must have already been initialized with `GrB_Matrix_new`, and it must have no entries in it before calling `GrB_Matrix_build`. Thus the dimensions and type of `C` are not changed by this function, but are inherited from the prior call to `GrB_Matrix_new` or `GrB_matrix_dup`.

An error is returned (`GrB_INDEX_OUT_OF_BOUNDS`) if any row index in `I` is greater than or equal to the number of rows of `C`, or if any column index in `J` is greater than or equal to the number of columns of `C`

Any duplicate entries with identical indices are assembled using the binary `dup` operator provided on input. All three types (`x`, `y`, `z` for `z=dup(x,y)`) must be identical. The types of `dup`, `C` and `X` must all be compatible. See Section 2.4 regarding typecasting and compatibility. The values in `X` are typecasted, if needed, into the type of `dup`. Duplicates are then assembled into a matrix `T` of the same type as `dup`, using `T(i,j) = dup (T (i,j), X (k))`. After `T` is constructed, it is typecasted into the result `C`. That is, typecasting does not occur at the same time as the assembly of duplicates.

If `dup` is `NULL`, any duplicates result in an error. If `dup` is the special binary operator `GxB_IGNORE_DUP`, then any duplicates are ignored. If duplicates appear, the last one in the list of tuples is taken and the prior ones ignored. This is not an error.

SPEC: As an extension to the specification, results are defined even if `dup` is non-associative.

The GraphBLAS API requires `dup` to be associative so that entries can be assembled in any order, and states that the result is undefined if `dup` is not associative. However, SuiteSparse:GraphBLAS guarantees a well-defined order of assembly. Entries in the tuples `[I,J,X]` are first sorted in increasing order of row and column index, with ties broken by the position of the tuple in the `[I,J,X]` list. If duplicates appear, they are assembled in the order they appear in the `[I,J,X]` input. That is, if the same indices `i` and `j` appear in positions `k1`, `k2`, `k3`, and `k4` in `[I,J,X]`, where `k1 < k2 < k3 < k4`, then the following operations will occur in order:

```
T (i,j) = X (k1) ;
T (i,j) = dup (T (i,j), X (k2)) ;
T (i,j) = dup (T (i,j), X (k3)) ;
T (i,j) = dup (T (i,j), X (k4)) ;
```

This is a well-defined order but the user should not depend upon it when using other GraphBLAS implementations since the GraphBLAS API does not require this ordering.

However, SuiteSparse:GraphBLAS guarantees this ordering, even when it compute the result in parallel. With this well-defined order, several operators become very useful. In particular, the `SECOND` operator results in the last tuple overwriting the earlier ones. The `FIRST` operator means the value of the first tuple is used and the others are discarded.

The acronym `dup` is used here for the name of binary function used for assembling duplicates, but this should not be confused with the `_dup` suffix in the name of the function `GrB_Matrix_dup`. The latter function does not apply any operator at all, nor any typecasting, but simply makes a pure deep copy of a matrix.

The parameter `X` is a pointer to any C equivalent built-in type, or a `void *` pointer. The `GrB_Matrix_build` function uses the `_Generic` feature of ANSI C11 to detect the type of pointer passed as the parameter `X`. If `X` is a pointer to a built-in type, then the function can do the right typecasting. If `X` is a `void *` pointer, then it can only assume `X` to be a pointer to a user-defined type that is the same user-defined type of `C` and `dup`. This function has no way of checking this condition that the `void * X` pointer points to an array of the correct user-defined type, so behavior is undefined if the user breaks this condition.

The `GrB_Matrix_build` method is analogous to `C = sparse (I,J,X)` in MATLAB, with several important extensions that go beyond that which MATLAB can do. In particular, the MATLAB `sparse` function only provides one option for assembling duplicates (summation), and it can only build double, double complex, and logical sparse matrices.

6.9.10 GxB_Matrix_build_Scalar: build a matrix from a set of tuples

```
GrB_Info GxB_Matrix_build_Scalar    // build a matrix from (I,J,scalar) tuples
(
    GrB_Matrix C,                   // matrix to build
```

```
    const GrB_Index *I,             // array of row indices of tuples
    const GrB_Index *J,             // array of column indices of tuples
    GrB_Scalar scalar,              // value for all tuples
    GrB_Index nvals                 // number of tuples
) ;
```

`GxB_Matrix_build_Scalar` constructs a sparse matrix `C` from a set of tuples defined the index arrays `I` and `J` of length `nvals`, and a scalar. The scalar is the value of all of the tuples. Unlike `GrB_Matrix_build`, there is no `dup` operator to handle duplicate entries. Instead, any duplicates are silently ignored (if the number of duplicates is desired, simply compare the input `nvals` with the value returned by `GrB_Vector_nvals` after the matrix is constructed). All entries in the sparsity pattern of `C` are identical, and equal to the input scalar value.

6.9.11 GrB_Matrix_setElement: add an entry to a matrix

```
GrB_Info GrB_Matrix_setElement          // C (i,j) = x
(
    GrB_Matrix C,                       // matrix to modify
    <type> x,                           // scalar to assign to C(i,j)
    GrB_Index i,                        // row index
    GrB_Index j                         // column index
) ;
```

`GrB_Matrix_setElement` sets a single entry in a matrix, `C(i,j)=x`. If the entry is already present in the pattern of `C`, it is overwritten with the new value. If the entry is not present, it is added to `C`. In either case, no entry is ever deleted by this function. Passing in a value of `x=0` simply creates an explicit entry at position `(i,j)` whose value is zero, even if the implicit value is assumed to be zero.

An error is returned (`GrB_INVALID_INDEX`) if the row index `i` is greater than or equal to the number of rows of `C`, or if the column index `j` is greater than or equal to the number of columns of `C`. Note that this error code differs from the same kind of condition in `GrB_Matrix_build`, which returns `GrB_INDEX_OUT_OF_BOUNDS`. This is because `GrB_INVALID_INDEX` is an API error, and is caught immediately even in non-blocking mode, whereas `GrB_INDEX_OUT_OF_BOUNDS` is an execution error whose detection may wait until the computation completes sometime later.

The scalar `x` is typecasted into the type of `C`. Any value can be passed to this function and its type will be detected, via the `_Generic` feature of ANSI C11. For a user-defined type, `x` is a `void *` pointer that points to a memory space holding a single entry of this user-defined type. This user-defined type must exactly match the user-defined type of `C` since no typecasting is done between user-defined types. If `x` is a `GrB_Scalar` and contains no entry, then the entry `C(i,j)` is removed (if it exists). The action taken is identical to `GrB_Matrix_removeElement(C,i,j)` in this case.

Performance considerations: SuiteSparse:GraphBLAS exploits the non-blocking mode to greatly improve the performance of this method. Refer to the example shown in Section 2.2. If the entry exists in the pattern already, it is updated right away and the work is not left pending. Otherwise, it is placed in a list of pending updates, and the later on the updates are done all at once, using the same algorithm used for `GrB_Matrix_build`. In other words, `setElement` in SuiteSparse:GraphBLAS builds its own internal list of tuples `[I,J,X]`, and then calls `GrB_Matrix_build` whenever the matrix is needed in another computation, or whenever `GrB_Matrix_wait` is called.

As a result, if calls to `setElement` are mixed with calls to most other methods and operations (even `extractElement`) then the pending updates are assembled right away, which will be slow. Performance will be good if many `setElement` updates are left pending, and performance will be poor if the updates are assembled frequently.

A few methods and operations can be intermixed with `setElement`, in particular, some forms of the `GrB_assign` and `GxB_subassign` operations are compatible with the pending updates from `setElement`. Section 10.11 gives more details on which `GxB_subassign` and `GrB_assign` operations can be interleaved with calls to `setElement` without forcing updates to be assembled. Other methods that do not access the existing entries may also

be done without forcing the updates to be assembled, namely `GrB_Matrix_clear` (which erases all pending updates), `GrB_Matrix_free`, `GrB_Matrix_ncols`, `GrB_Matrix_nrows`, `GxB_Matrix_type`, and of course `GrB_Matrix_setElement` itself. All other methods and operations cause the updates to be assembled. Future versions of SuiteSparse:GraphBLAS may extend this list.

See Section 15.2 for an example of how to use `GrB_Matrix_setElement`. If an error occurs, `GrB_error(&err,C)` returns details about the error.

6.9.12 GrB_Matrix_extractElement: get an entry from a matrix

```
GrB_Info GrB_Matrix_extractElement      // x = A(i,j)
(
    <type> *x,                  // extracted scalar (non-opaque C scalar)
    const GrB_Matrix A,         // matrix to extract a scalar from
    GrB_Index i,                // row index
    GrB_Index j                 // column index
) ;

GrB_Info GrB_Matrix_extractElement      // x = A(i,j)
(
    GrB_Scalar x,               // extracted GrB_Scalar
    const GrB_Matrix A,         // matrix to extract a scalar from
    GrB_Index i,                // row index
    GrB_Index j                 // column index
) ;
```

`GrB_Matrix_extractElement` extracts a single entry from a matrix `x=A(i,j)`.

An error is returned (`GrB_INVALID_INDEX`) if the row index `i` is greater than or equal to the number of rows of `C`, or if column index `j` is greater than or equal to the number of columns of `C`.

If the entry is present, `x=A(i,j)` is performed and the scalar `x` is returned with this value. The method returns `GrB_SUCCESS`.

If no entry is present at `A(i,j)`, and `x` is a non-opaque C scalar, then `x` is not modified, and the return value of `GrB_Matrix_extractElement` is `GrB_NO_VALUE`. If `x` is a `GrB_Scalar`, then `x` is returned as an empty scalar with no entry, and `GrB_SUCCESS` is returned.

The function knows the type of the pointer `x`, so it can do typecasting as needed, from the type of `A` into the type of `x`. User-defined types cannot be typecasted, so if `A` has a user-defined type then `x` must be a `void *` pointer that points to a memory space the same size as a single scalar of the type of `A`.

Currently, this method causes all pending updates from `GrB_setElement`, `GrB_assign`, or `GxB_subassign` to be assembled, so its use can have performance implications. Calls to this function should not be arbitrarily intermixed with calls to these other two functions. Everything will work correctly and results will be predictable, it will just be slow.

6.9.13 GrB_Matrix_removeElement: remove an entry from a matrix

```
GrB_Info GrB_Matrix_removeElement
(
    GrB_Matrix C,                   // matrix to remove an entry from
    GrB_Index i,                    // row index
    GrB_Index j                     // column index
) ;
```

`GrB_Matrix_removeElement` removes a single entry `A(i,j)` from a matrix. If no entry is present at `A(i,j)`, then the matrix is not modified. If an error occurs, `GrB_error(&err,A)` returns details about the error.

6.9.14 GrB_Matrix_extractTuples: get all entries from a matrix

```
GrB_Info GrB_Matrix_extractTuples           // [I,J,X] = find (A)
(
    GrB_Index *I,                 // array for returning row indices of tuples
    GrB_Index *J,                 // array for returning col indices of tuples
    <type> *X,                    // array for returning values of tuples
    GrB_Index *nvals,             // I,J,X size on input; # tuples on output
    const GrB_Matrix A            // matrix to extract tuples from
) ;
```

`GrB_Matrix_extractTuples` extracts all the entries from the matrix `A`, returning them as a list of tuples, analogous to `[I,J,X]=find(A)` in MATLAB. Entries in the tuples `[I,J,X]` are unique. No pair of row and column indices `(i,j)` appears more than once.

The GraphBLAS API states the tuples can be returned in any order. If `GrB_wait` is called first, then SuiteSparse:GraphBLAS chooses to always return them in sorted order, depending on whether the matrix is stored by row or by column. Otherwise, the indices can be returned in any order.

The number of tuples in the matrix `A` is given by `GrB_Matrix_nvals(&anvals,A)`. If `anvals` is larger than the size of the arrays (`nvals` in the parameter list), an error `GrB_INSUFFICIENT_SIZE` is returned, and no tuples are extracted. If `nvals` is larger than `anvals`, then only the first `anvals` entries in the arrays `I J`, and `X` are modified, containing all the tuples of `A`, and the rest of `I J`, and `X` are left unchanged. On output, `nvals` contains the number of tuples extracted.

SPEC: As an extension to the specification, the arrays `I`, `J`, and/or `X` may be passed in as `NULL` pointers. `GrB_Matrix_extractTuples` does not return a component specified as `NULL`. This is not an error condition.

6.9.15 GrB_Matrix_resize: resize a matrix

```
GrB_Info GrB_Matrix_resize      // change the size of a matrix
(
    GrB_Matrix A,               // matrix to modify
    const GrB_Index nrows_new,  // new number of rows in matrix
    const GrB_Index ncols_new   // new number of columns in matrix
) ;
```

`GrB_Matrix_resize` changes the size of a matrix. If the dimensions decrease, entries that fall outside the resized matrix are deleted.

6.9.16 GxB_Matrix_concat: concatenate matrices

```
GrB_Info GxB_Matrix_concat          // concatenate a 2D array of matrices
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix *Tiles,        // 2D row-major array of size m-by-n
    const GrB_Index m,
    const GrB_Index n,
    const GrB_Descriptor desc       // unused, except threading control
) ;
```

`GxB_Matrix_concat` concatenates an array of matrices (`Tiles`) into a single `GrB_Matrix C`.

`Tiles` is an `m`-by-`n` dense array of matrices held in row-major format, where `Tiles [i*n+j]` is the (i, j)th tile, and where `m` > 0 and `n` > 0 must hold. Let $A_{i,j}$ denote the (i, j)th tile. The matrix `C` is constructed by concatenating these tiles together, as:

$$C = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & A_{1,2} & \cdots & A_{1,n-1} \\ \cdots & & & & \\ A_{m-1,0} & A_{m-1,1} & A_{m-1,2} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

On input, the matrix `C` must already exist. Any existing entries in `C` are discarded. `C` must have dimensions `nrows` by `ncols` where `nrows` is the sum of the number of rows in the matrices $A_{i,0}$ for all i, and `ncols` is the sum of the number of columns in the matrices $A_{0,j}$ for all j. All matrices in any given tile row i must have the same number of rows (that is, and all matrices in any given tile column j must have the same number of columns).

The type of `C` is unchanged, and all matrices $A_{i,j}$ are typecasted into the type of `C`. Any settings made to `C` by `GxB_Matrix_Option_set` (format by row or by column, bitmap switch, hyper switch, and sparsity control) are unchanged.

6.9.17 GxB_Matrix_split: split a matrix

```
GrB_Info GxB_Matrix_split           // split a matrix into 2D array of matrices
(
    GrB_Matrix *Tiles,              // 2D row-major array of size m-by-n
    const GrB_Index m,
    const GrB_Index n,
    const GrB_Index *Tile_nrows,    // array of size m
    const GrB_Index *Tile_ncols,    // array of size n
    const GrB_Matrix A,             // input matrix to split
    const GrB_Descriptor desc       // unused, except threading control
) ;
```

`GxB_Matrix_split` does the opposite of `GxB_Matrix_concat`. It splits a single input matrix `A` into a 2D array of tiles. On input, the `Tiles` array must be a non-`NULL` pointer to a previously allocated array of size at least `m*n` where both `m` and `n` must be greater than zero. The `Tiles_nrows` array has size `m`, and `Tiles_ncols` has size `n`. The (i, j)th tile has dimension `Tiles_nrows[i]`-by-`Tiles_ncols[j]`. The sum of `Tiles_nrows [0:m-1]` must equal the number of rows of `A`, and the sum of `Tiles_ncols [0:n-1]` must equal the number of columns of `A`. The type of each tile is the same as the type of `A`; no typecasting is done.

6.9.18 GrB_Matrix_diag: construct a diagonal matrix

```
GrB_Info GrB_Matrix_diag    // construct a diagonal matrix from a vector
(
    GrB_Matrix C,                   // output matrix
    const GrB_Vector v,             // input vector
    int64_t k
) ;
```

`GrB_Matrix_diag` constructs a matrix from a vector. Let n be the length of the `v` vector, from `GrB_Vector_size (&n, v)`. If `k` $= 0$, then `C` is an n-by-n diagonal matrix with the entries from `v` along the main diagonal of `C`, with `C(i,i)=v(i)`. If `k` is nonzero, `C` is square with dimension $n + |k|$. If `k` is positive, it denotes diagonals above the main diagonal, with `C(i,i+k)=v(i)`. If `k` is negative, it denotes diagonals below the main diagonal of `C`, with `C(i-k,i)=v(i)`. This behavior is identical to the MATLAB statement `C=diag(v,k)`, where `v` is a vector, except that `GrB_Matrix_diag` can also do typecasting.

`C` must already exist on input, of the correct size. Any existing entries in `C` are discarded. The type of `C` is preserved, so that if the type of `C` and `v` differ, the entries are typecasted into the type of `C`. Any settings made to `C` by `GxB_Matrix_Option_set` (format by row or by column, bitmap switch, hyper switch, and sparsity control) are unchanged.

6.9.19 GxB_Matrix_diag: construct a diagonal matrix

```
GrB_Info GxB_Matrix_diag    // construct a diagonal matrix from a vector
(
    GrB_Matrix C,                   // output matrix
    const GrB_Vector v,             // input vector
    int64_t k,
    const GrB_Descriptor desc       // unused, except threading control
) ;
```

Identical to `GrB_Matrix_diag`, except for the extra parameter: a `descriptor` to provide control over the number of threads used.

6.9.20 GxB_Matrix_iso: query iso status of a matrix

```
GrB_Info GxB_Matrix_iso     // return iso status of a matrix
(
    bool *iso,              // true if the matrix is iso-valued
    const GrB_Matrix A      // matrix to query
) ;
```

Returns the true if the matrix is iso-valued, false otherwise.

6.9.21 GxB_Matrix_memoryUsage: memory used by a matrix

```
GrB_Info GxB_Matrix_memoryUsage  // return # of bytes used for a matrix
(
    size_t *size,           // # of bytes used by the matrix A
    const GrB_Matrix A      // matrix to query
) ;
```

Returns the memory space required for a matrix, in bytes.

6.9.22 GrB_Matrix_free: free a matrix

```
GrB_Info GrB_free           // free a matrix
(
    GrB_Matrix *A           // handle of matrix to free
) ;
```

`GrB_Matrix_free` frees a matrix. Either usage:

```
GrB_Matrix_free (&A) ;
GrB_free (&A) ;
```

frees the matrix `A` and sets `A` to `NULL`. It safely does nothing if passed a `NULL` handle, or if `A == NULL` on input. Any pending updates to the matrix are abandoned.

6.10 Serialize/deserialize methods

Serialization takes an opaque GraphBLAS object (a vector or matrix) and encodes it in a single non-opaque array of bytes, the *blob*. The blob can only be deserialized by the same library that created it (SuiteSparse:GraphBLAS in this case). The array of bytes can be written to a file, sent to another process over an MPI channel, or operated on in any other way that moves the bytes around. The contents of the array cannot be interpreted except by deserialization back into a vector or matrix, by the same library (and sometimes the same version) that created the blob. Currently, all versions of SuiteSparse:GraphBLAS that implement serialization/deserialization use the same format for the blob, so the library versions are compatible with each other.

There are two forms of serialization: `GrB*serialize` and `GxB*serialize`. For the `GrB` form, the blob must first be allocated by the user application, and it must be large enough to hold the matrix or vector.

By default, LZ4 compression is used for serialization, but other options can be selected via the descriptor: `GxB_set (desc, GxB_COMPRESSION, method)`, where `method` is an integer selected from the following options:

method	description
`GxB_COMPRESSION_NONE`	no compression
`GxB_COMPRESSION_DEFAULT`	LZ4
`GxB_COMPRESSION_LZ4`	LZ4
`GxB_COMPRESSION_LZ4HC`	LZ4HC, with default level 9

The LZ4HC method can be modified by adding a level of zero to 9, with 9 being the default. Higher levels lead to a more compact blob, at the cost of extra computational time. This level is simply added to the method, so to compress a vector with LZ4HC with level 6, use:

```
GxB_set (desc, GxB_COMPRESSION, GxB_COMPRESSION_LZ4HC + 6) ;
```

Deserialization of untrusted data is a common security problem; see https://cwe.mitre.org/data/definitions/502.html. The deserialization methods do a few basic checks so that no out-of-bounds access occurs during deserialization, but the output matrix or vector itself may still be corrupted. If the data is untrusted, use check the matrix or vector after deserializing it:

```
info = GxB_Vector_fprint (w, "w deserialized", GrB_SILENT, NULL) ;
if (info != GrB_SUCCESS) GrB_free (&w) ;
info = GxB_Matrix_fprint (A, "A deserialized", GrB_SILENT, NULL) ;
if (info != GrB_SUCCESS) GrB_free (&A) ;
```

The following methods are described in this Section:

GraphBLAS function	purpose	Section
`GxB_Vector_serialize`	serialize a vector	6.10.1
`GxB_Vector_deserialize`	deserialize a vector	6.10.2
`GrB_Matrix_serializeSize`	return size of serialized matrix	6.10.3
`GrB_Matrix_serialize`	serialize a matrix	6.10.4
`GxB_Matrix_serialize`	serialize a matrix	6.10.5
`GrB_Matrix_deserialize`	deserialize a matrix	6.10.6
`GxB_Matrix_deserialize`	deserialize a matrix	6.10.7
`GrB_deserialize_type_name`	return the name of type of the blob	6.10.8

6.10.1 GxB_Vector_serialize: serialize a vector

```
GrB_Info GxB_Vector_serialize       // serialize a GrB_Vector to a blob
(
    // output:
    void **blob_handle,             // the blob, allocated on output
    GrB_Index *blob_size_handle,    // size of the blob on output
    // input:
    GrB_Vector u,                   // vector to serialize
    const GrB_Descriptor desc       // descriptor to select compression method
                                    // and to control # of threads used
) ;
```

`GxB_Vector_serialize` serializes a vector into a single array of bytes (the blob), which is `malloc`'ed and filled with the serialized vector. By default, LZ4 compression is used, but other options can be selected via the descriptor. Serializing a vector is identical to serializing a matrix; see Section 6.10.5 for more information.

6.10.2 GxB_Vector_deserialize: deserialize a vector

```
GrB_Info GxB_Vector_deserialize     // deserialize blob into a GrB_Vector
(
    // output:
    GrB_Vector *w,      // output vector created from the blob
    // input:
    GrB_Type type,      // type of the vector w.  See GxB_Matrix_deserialize.
    const void *blob,       // the blob
    GrB_Index blob_size,    // size of the blob
    const GrB_Descriptor desc       // to control # of threads used
) ;
```

This method creates a vector `w` by deserializing the contents of the blob, constructed by `GxB_Vector_serialize`. Deserializing a vector is identical to deserializing a matrix; see Section 6.10.7 for more information.

6.10.3 GrB_Matrix_serializeSize: return size of serialized matrix

```
GrB_Info GrB_Matrix_serializeSize   // estimate the size of a blob
(
    // output:
    GrB_Index *blob_size_handle,    // upper bound on the required size of the
                                    // blob on output.
    // input:
    GrB_Matrix A                    // matrix to serialize
) ;
```

`GrB_Matrix_serializeSize` returns an upper bound on the size of the blob needed to serialize a `GrB_Matrix` with `GrB_Matrix_serialize`. After the matrix is serialized, the actual size used is returned, and the blob may be `realloc`'d to that size if desired. This method is not required for `GxB_Matrix_serialize`.

6.10.4 GrB_Matrix_serialize: serialize a matrix

```
GrB_Info GrB_Matrix_serialize       // serialize a GrB_Matrix to a blob
(
    // output:
    void *blob,                     // the blob, already allocated in input
    // input/output:
    GrB_Index *blob_size_handle,    // size of the blob on input.  On output,
                                    // the # of bytes used in the blob.
    // input:
    GrB_Matrix A                    // matrix to serialize
) ;
```

`GrB_Matrix_serialize` serializes a matrix into a single array of bytes (the blob), which must be already allocated by the user application. On input, `&blob_size` is the size of the allocated blob in bytes. On output, it is reduced to the numbed of bytes actually used to serialize the matrix. After calling `GrB_Matrix_serialize`, the blob may be `realloc`'d to this revised size if desired (this is optional). LZ4 compression is used to construct a compact blob.

6.10.5 GxB_Matrix_serialize: serialize a matrix

```
GrB_Info GxB_Matrix_serialize       // serialize a GrB_Matrix to a blob
(
    // output:
    void **blob_handle,             // the blob, allocated on output
    GrB_Index *blob_size_handle,    // size of the blob on output
    // input:
    GrB_Matrix A,                   // matrix to serialize
    const GrB_Descriptor desc       // descriptor to select compression method
                                    // and to control # of threads used
) ;
```

`GxB_Matrix_serialize` is identical to `GrB_Matrix_serialize`, except that it does not require a pre-allocated blob. Instead, it `malloc`'s the blob internally, and fills it with the serialized matrix. By default, LZ4 compression is used, but other options can be selected via the descriptor.

6.10.6 GrB_Matrix_deserialize: deserialize a matrix

```
GrB_Info GrB_Matrix_deserialize     // deserialize blob into a GrB_Matrix
(
    // output:
    GrB_Matrix *C,      // output matrix created from the blob
    // input:
    GrB_Type type,      // type of the matrix C.  Required if the blob holds a
                        // matrix of user-defined type.  May be NULL if blob
                        // holds a built-in type; otherwise must match the
                        // type of C.
    const void *blob,       // the blob
    GrB_Index blob_size     // size of the blob
) ;
```

This method creates a matrix `A` by deserializing the contents of the blob, constructed by either `GrB_Matrix_serialize` or `GxB_Matrix_serialize`.

SPEC: The specification requires the `type` to always be non-NULL. As an extension, SuiteSparse:GraphBLAS allows `type` to be NULL if the blob contains a serialized matrix with a built-in type.

6.10.7 GxB_Matrix_deserialize: deserialize a matrix

```
GrB_Info GxB_Matrix_deserialize     // deserialize blob into a GrB_Matrix
(
    // output:
    GrB_Matrix *C,      // output matrix created from the blob
    // input:
    GrB_Type type,      // type of the matrix C.  Required if the blob holds a
                        // matrix of user-defined type.  May be NULL if blob
                        // holds a built-in type; otherwise must match the
                        // type of C.
    const void *blob,       // the blob
    GrB_Index blob_size,    // size of the blob
    const GrB_Descriptor desc       // to control # of threads used
) ;
```

Identical to `GrB_Matrix_deserialize`, except that the descriptor appears as the last parameter to control the number of threads used.

6.10.8 GxB_deserialize_type_name: name of the type of a blob

```
GrB_Info GxB_deserialize_type_name  // return the type name of a blob
(
    // output:
    char *type_name,        // name of the type (char array of size at least
                            // GxB_MAX_NAME_LEN, owned by the user application).
    // input, not modified:
    const void *blob,       // the blob
    GrB_Index blob_size     // size of the blob
) ;
```

`GrB_deserialize_type_name` returns the name of type of the matrix or vector serialized into the blob. This method works for any blob, from `GxB_Vector_serialize`, `GrB_Matrix_serialize`, or `GxB_Matrix_serialize`.

6.11 GraphBLAS pack/unpack: using move semantics

The pack/unpack functions allow the user application to create a `GrB_Matrix` or `GrB_Vector` object, and to extract its contents, faster and with less memory overhead than the `GrB_*_build` and `GrB_*_extractTuples` functions.

The `GrB_Matrix_import` and `GrB_Matrix_export` are not described in this section. Refer to Section 6.12 instead.

The semantics of the `GxB` pack/unpack are the same as the *move constructor* in C++. For `GxB*pack*`, the user provides a set of arrays that have been previously allocated via the ANSI C `malloc`, `calloc`, or `realloc` functions (by default), or by the corresponding functions passed to `GxB_init`. The arrays define the content of the matrix or vector. Unlike `GrB_*_build`, the GraphBLAS library then takes ownership of the user's input arrays and may either:

1. incorporate them into its internal data structure for the new `GrB_Matrix` or `GrB_Vector`, potentially creating the `GrB_Matrix` or `GrB_Vector` in constant time with no memory copying performed, or
2. if the library does not support the format directly, then it may convert the input to its internal format, and then free the user's input arrays.
3. A GraphBLAS implementation may also choose to use a mix of the two strategies.

SuiteSparse:GraphBLAS takes the first approach, and so the pack functions always take $O(1)$ time, and require $O(1)$ memory space to be allocated.

Regardless of the method chosen, as listed above, the input arrays are no longer owned by the user application. If `A` is a `GrB_Matrix` created by a pack method, the user input arrays are freed no later than `GrB_free(&A)`, and may be freed earlier, at the discretion of the GraphBLAS library. The data structure of the `GrB_Matrix` and `GrB_Vector` remain opaque.

The `GxB*unpack*` of a `GrB_Matrix` or `GrB_Vector` is symmetric with the pack operation. The unpack changes the ownership of the arrays, which are returned to the user and which contain the matrix or vector in the requested format. Ownership of these arrays is given to the user application, which is then responsible for freeing them via the ANSI C `free` function (by default), or by the `free_function` that was passed in to `GxB_init`. Alternatively, these arrays can be re-packed into a `GrB_Matrix` or `GrB_Vector`, at which point they again become the responsibility of GraphBLAS.

For an unpack method, if the output format matches the current internal format of the matrix or vector then these arrays are returned to the user application in $O(1)$ time and with no memory copying performed. Otherwise, the `GrB_Matrix` or `GrB_Vector` is first converted into the requested format, and then unpacked.

For the pack methods, the `A` matrix/vector must already exist on input, and its contents are populated with the new content, just like `GrB_Matrix_build`. For the unpack methods, `A` is passed in, and the matrix/vector still exists on return, just with no entries. Its type and dimensions are preserved.

Unpacking a matrix or vector forces completion of any pending operations on the matrix, with one exception. SuiteSparse:GraphBLAS supports three kinds of pending operations: *zombies* (pending deletions), *pending tuples* (pending insertions), and a *lazy*

sort. Zombies and pending tuples are never unpacked, but the *jumbled* state may be optionally unpacked. In the latter, if the matrix or vector is unpacked in a *jumbled* state, indices in any row or column may appear out of order. If unpacked as *unjumbled*, the indices always appear in ascending order.

The vector pack/unpack methods use three formats for a `GrB_Vector`. Eight different formats are provided for the pack/unpack of a `GrB_Matrix`. For each format, the numerical value array (`Ax` or `vx`) has a C type corresponding to one of the 13 built-in types in GraphBLAS (`bool`, `int*_t`, `uint*_t`, `float`, `double` `float complex`, `double complex`), or that corresponds with the user-defined type. No typecasting is done.

If `iso` is true, then all entries present in the matrix or vector have the same value, and the `Ax` array (for matrices) or `vx` array (for vectors) only need to be large enough to hold a single value.

The unpack of a `GrB_Vector` in `CSC` format may return the indices in a jumbled state, in any order. For a `GrB_Matrix` in `CSR` or `HyperCSR` format, if the matrix is returned as jumbled, the column indices in any given row may appear out of order. For `CSC` or `HyperCSC` formats, if the matrix is returned as jumbled, the row indices in any given column may appear out of order.

On pack, if the user-provided arrays contain jumbled row or column vectors, then the input flag `jumbled` must be passed in as `true`. On unpack, if `*jumbled` is `NULL`, this indicates to the unpack method that the user expects the unpacked matrix or vector to be returned in an ordered, unjumbled state. If `*jumbled` is provided as non-`NULL`, then it is returned as `true` if the indices may appear out of order, or `false` if they are known to be in ascending order.

Matrices and vectors in bitmap or full format are never jumbled.

If data is packed using `GxB*_pack_*`, the default is to trust the input data so that the pack can be done in $O(1)$ time. However, if the data comes from an untrusted source, additional checks should be made during the pack. This is indicated with a descriptor setting, and then passing the descriptor to the `GxB` pack methods:

```
GxB_set (desc, GxB_IMPORT, GxB_SECURE_IMPORT) ;
```

The table below lists the methods presented in this section.

method	purpose	Section
GxB_Vector_pack_CSC	pack a vector in CSC format	6.11.1
GxB_Vector_unpack_CSC	unpack a vector in CSC format	6.11.2
GxB_Vector_pack_Bitmap	pack a vector in bitmap format	6.11.3
GxB_Vector_unpack_Bitmap	unpack a vector in bitmap format	6.11.4
GxB_Vector_pack_Full	pack a vector in full format	6.11.5
GxB_Vector_unpack_Full	unpack a vector in full format	6.11.6
GxB_Matrix_pack_CSR	pack a matrix in CSR form	6.11.7
GxB_Matrix_unpack_CSR	unpack a matrix in CSR form	6.11.8
GxB_Matrix_pack_CSC	pack a matrix in CSC form	6.11.9
GxB_Matrix_unpack_CSC	unpack a matrix in CSC form	6.11.10
GxB_Matrix_pack_HyperCSR	pack a matrix in HyperCSR form	6.11.11
GxB_Matrix_unpack_HyperCSR	unpack a matrix in HyperCSR form	6.11.12
GxB_Matrix_pack_HyperCSC	pack a matrix in HyperCSC form	6.11.13
GxB_Matrix_unpack_HyperCSC	unpack a matrix in HyperCSC form	6.11.14
GxB_Matrix_pack_BitmapR	pack a matrix in BitmapR form	6.11.15
GxB_Matrix_unpack_BitmapR	unpack a matrix in BitmapR form	6.11.16
GxB_Matrix_pack_BitmapC	pack a matrix in BitmapC form	6.11.17
GxB_Matrix_unpack_BitmapC	unpack a matrix in BitmapC form	6.11.18
GxB_Matrix_pack_FullR	pack a matrix in FullR form	6.11.19
GxB_Matrix_unpack_FullR	unpack a matrix in FullR form	6.11.20
GxB_Matrix_pack_FullC	pack a matrix in FullC form	6.11.21
GxB_Matrix_unpack_FullC	unpack a matrix in FullC form	6.11.22

6.11.1 GxB_Vector_pack_CSC pack a vector in CSC form

```
GrB_Info GxB_Vector_pack_CSC  // pack a vector in CSC format
(
    GrB_Vector v,       // vector to create (type and length unchanged)
    GrB_Index **vi,     // indices, vi_size >= nvals(v) * sizeof(int64_t)
    void **vx,          // values, vx_size >= nvals(v) * (type size)
                        // or vx_size >= (type size), if iso is true
    GrB_Index vi_size,  // size of vi in bytes
    GrB_Index vx_size,  // size of vx in bytes
    bool iso,           // if true, v is iso
    GrB_Index nvals,    // # of entries in vector
    bool jumbled,       // if true, indices may be unsorted
    const GrB_Descriptor desc
) ;
```

`GxB_Vector_pack_CSC` is analogous to `GxB_Matrix_pack_CSC`. Refer to the description of `GxB_Matrix_pack_CSC` for details (Section 6.11.9).

The vector `v` must exist on input with the right type and length. No typecasting is done. Its entries are the row indices given by `vi`, with the corresponding values in `vx`. The two pointers `vi` and `vx` are returned as `NULL`, which denotes that they are no longer owned by the user application. They have instead been moved into `v`. If `jumbled` is true, the row indices in `vi` must appear in sorted order. No duplicates can appear. These conditions are not checked, so results are undefined if they are not met exactly. The user application can check the resulting vector `v` with `GxB_print`, if desired, which will determine if these conditions hold.

If not successful, `v`, `vi` and `vx` are not modified.

6.11.2 GxB_Vector_unpack_CSC: unpack a vector in CSC form

```
GrB_Info GxB_Vector_unpack_CSC  // unpack a CSC vector
(
    GrB_Vector v,       // vector to unpack (type and length unchanged)
    GrB_Index **vi,     // indices
    void **vx,          // values
    GrB_Index *vi_size, // size of vi in bytes
    GrB_Index *vx_size, // size of vx in bytes
    bool *iso,          // if true, v is iso
    GrB_Index *nvals,   // # of entries in vector
    bool *jumbled,      // if true, indices may be unsorted
    const GrB_Descriptor desc
) ;
```

`GxB_Vector_unpack_CSC` is analogous to `GxB_Matrix_unpack_CSC`. Refer to the description of `GxB_Matrix_unpack_CSC` for details (Section 6.11.10).

Exporting a vector forces completion of any pending operations on the vector, except that indices may be unpacked out of order (`jumbled` is `true` if they may be out of order, `false` if sorted in ascending order). If `jumbled` is `NULL` on input, then the indices are always returned in sorted order.

If successful, `v` is returned with no entries, and its contents are returned to the user. A list of row indices of entries that were in `v` is returned in `vi`, and the corresponding numerical values are returned in `vx`. If `nvals` is zero, the `vi` and `vx` arrays are returned as `NULL`; this is not an error condition.

If not successful, `v` is unmodified and `vi` and `vx` are not modified.

6.11.3 GxB_Vector_pack_Bitmap pack a vector in bitmap form

```
GrB_Info GxB_Vector_pack_Bitmap // pack a bitmap vector
(
    GrB_Vector v,       // vector to create (type and length unchanged)
    int8_t **vb,        // bitmap, vb_size >= n
    void **vx,          // values, vx_size >= n * (type size)
                        // or vx_size >= (type size), if iso is true
    GrB_Index vb_size,  // size of vb in bytes
    GrB_Index vx_size,  // size of vx in bytes
    bool iso,           // if true, v is iso
    GrB_Index nvals,    // # of entries in bitmap
    const GrB_Descriptor desc
) ;
```

`GxB_Vector_pack_Bitmap` is analogous to `GxB_Matrix_pack_BitmapC`. Refer to the description of `GxB_Matrix_pack_BitmapC` for details (Section 6.11.17).

The vector `v` must exist on input with the right type and length. No typecasting is done. Its entries are determined by `vb`, where `vb[i]=1` denotes that the entry $v(i)$ is present with value given by `vx[i]`, and `vb[i]=0` denotes that the entry $v(i)$ is not present (`vx[i]` is ignored in this case).

The two pointers `vb` and `vx` are returned as `NULL`, which denotes that they are no longer owned by the user application. They have instead been moved into the new `GrB_Vector v`.

The `vb` array must not hold any values other than 0 and 1. The value `nvals` must exactly match the number of 1s in the `vb` array. These conditions are not checked, so results are undefined if they are not met exactly. The user application can check the resulting vector `v` with `GxB_print`, if desired, which will determine if these conditions hold.

If not successful, `v`, `vb` and `vx` are not modified.

6.11.4 GxB_Vector_unpack_Bitmap: unpack a vector in bitmap form

```
GrB_Info GxB_Vector_unpack_Bitmap   // unpack a bitmap vector
(
    GrB_Vector v,       // vector to unpack (type and length unchanged)
    int8_t **vb,        // bitmap
    void **vx,          // values
    GrB_Index *vb_size, // size of vb in bytes
    GrB_Index *vx_size, // size of vx in bytes
    bool *iso,          // if true, v is iso
    GrB_Index *nvals,    // # of entries in bitmap
    const GrB_Descriptor desc
) ;
```

`GxB_Vector_unpack_Bitmap` is analogous to `GxB_Matrix_unpack_BitmapC`; see Section 6.11.18. Exporting a vector forces completion of any pending operations on the vector. If successful, `v` is returned with no entries, and its contents are returned to the user. The entries that were in `v` are returned in `vb`, where `vb[i]=1` means $v(i)$ is present with value `vx[i]`, and `vb[i]=0` means $v(i)$ is not present (`vx[i]` is undefined in this case). The corresponding numerical values are returned in `vx`.

If not successful, `v` is unmodified and `vb` and `vx` are not modified.

6.11.5 GxB_Vector_pack_Full pack a vector in full form

```
GrB_Info GxB_Vector_pack_Full // pack a full vector
(
    GrB_Vector v,       // vector to create (type and length unchanged)
    void **vx,          // values, vx_size >= nvals(v) * (type size)
                        // or vx_size >= (type size), if iso is true
    GrB_Index vx_size,  // size of vx in bytes
    bool iso,           // if true, v is iso
    const GrB_Descriptor desc
) ;
```

`GxB_Vector_pack_Full` is analogous to `GxB_Matrix_pack_FullC`. Refer to the description of `GxB_Matrix_pack_BitmapC` for details (Section 6.11.21). The vector `v` must exist on input with the right type and length. No typecasting is done. If successful, `v` has all entries are present, and the value of $v(i)$ is given by `vx[i]`. The pointer `vx` is returned as `NULL`, which denotes that it is no longer owned by the user application. It has instead been moved into the new `GrB_Vector v`. If not successful, `v` and `vx` are not modified.

6.11.6 GxB_Vector_unpack_Full: unpack a vector in full form

```
GrB_Info GxB_Vector_unpack_Full   // unpack a full vector
(
    GrB_Vector v,       // vector to unpack (type and length unchanged)
    void **vx,          // values
    GrB_Index *vx_size, // size of vx in bytes
    bool *iso,          // if true, v is iso
    const GrB_Descriptor desc
) ;
```

`GxB_Vector_unpack_Full` is analogous to `GxB_Matrix_unpack_FullC`. Refer to the description of `GxB_Matrix_unpack_FullC` for details (Section 6.11.22). Exporting a vector forces completion of any pending operations on the vector. All entries in `v` must be present. In other words, prior to the unpack, `GrB_Vector_nvals` for a vector of length `n` must report that the vector contains `n` entries; `GrB_INVALID_VALUE` is returned if this condition does not hold. If successful, `v` is returned with no entries, and its contents are returned to the user. The entries that were in `v` are returned in the array `vx`, `vb`, where `vb[i]=1` means $v(i)$ is present with value where the value of $v(i)$ is `vx[i]`. If not successful, `v` and `vx` are not modified.

6.11.7 GxB_Matrix_pack_CSR: pack a CSR matrix

```
GrB_Info GxB_Matrix_pack_CSR      // pack a CSR matrix
(
    GrB_Matrix A,       // matrix to create (type, nrows, ncols unchanged)
    GrB_Index **Ap,     // row "pointers", Ap_size >= (nrows+1)* sizeof(int64_t)
    GrB_Index **Aj,     // column indices, Aj_size >= nvals(A) * sizeof(int64_t)
    void **Ax,          // values, Ax_size >= nvals(A) * (type size)
                        // or Ax_size >= (type size), if iso is true
    GrB_Index Ap_size,  // size of Ap in bytes
    GrB_Index Aj_size,  // size of Aj in bytes
    GrB_Index Ax_size,  // size of Ax in bytes
    bool iso,           // if true, A is iso
    bool jumbled,       // if true, indices in each row may be unsorted
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_pack_CSR` packs a matrix from 3 user arrays in CSR format. In the resulting `GrB_Matrix A`, the `CSR` format is a sparse matrix with a format (`GxB_FORMAT`) of `GxB_BY_ROW`.

The `GrB_Matrix A` must exist on input with the right type and dimensions. No typecasting is done.

This function populates the matrix `A` with the three arrays `Ap`, `Aj` and `Ax`, provided by the user, all of which must have been created with the ANSI C `malloc`, `calloc`, or `realloc` functions (by default), or by the corresponding `malloc_function`, `calloc_function`, or `realloc_function` provided to `GxB_init`. These arrays define the pattern and values of the new matrix `A`:

- `GrB_Index Ap [nrows+1] ;` The `Ap` array is the row "pointer" array. It does not actual contain pointers. More precisely, it is an integer array that defines where the column indices and values appear in `Aj` and `Ax`, for each row. The number of entries in row `i` is given by the expression `Ap [i+1] - Ap [i]`.
- `GrB_Index Aj [nvals] ;` The `Aj` array defines the column indices of entries in each row.
- `ctype Ax [nvals] ;` The `Ax` array defines the values of entries in each row. It is passed in as a `(void *)` pointer, but it must point to an array of size `nvals` values, each of size `sizeof(ctype)`, where `ctype` is the exact type in C that corresponds to the `GrB_Type type` parameter. That is, if `type` is `GrB_INT32`, then `ctype` is `int32_t`. User types may be used, just the same as built-in types.

The content of the three arrays `Ap Aj`, and `Ax` is very specific. This content is not checked, since this function takes only $O(1)$ time. Results are undefined if the following specification is not followed exactly.

The column indices of entries in the ith row of the matrix are held in `Aj [Ap [i] ... Ap[i+1]]`, and the corresponding values are held in the same positions in `Ax`. Column indices must be

in the range 0 to `ncols`-1. If `jumbled` is `false`, column indices must appear in ascending order within each row. If `jumbled` is `true`, column indices may appear in any order within each row. No duplicate column indices may appear in any row. `Ap [0]` must equal zero, and `Ap [nrows]` must equal nvals. The `Ap` array must be of size `nrows`+1 (or larger), and the `Aj` and `Ax` arrays must have size at least `nvals`.

If `nvals` is zero, then the content of the `Aj` and `Ax` arrays is not accessed and they may be `NULL` on input (if not `NULL`, they are still freed and returned as `NULL`, if the method is successful).

An example of the CSR format is shown below. Consider the following matrix with 10 nonzero entries, and suppose the zeros are not stored.

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix} \tag{1}$$

The `Ap` array has length 5, since the matrix is 4-by-4. The first entry must always zero, and `Ap [5] = 10` is the number of entries. The content of the arrays is shown below:

```
int64_t Ap [ ] = { 0,        2,              5,        7,            10 } ;
int64_t Aj [ ] = { 0,   2,   0,   1,   3,   1,   2,   0,   1,   3   } ;
double  Ax [ ] = { 4.5, 3.2, 3.1, 2.9, 0.9, 1.7, 3.0, 3.5, 0.4, 1.0 } ;
```

Spaces have been added to the `Ap` array, just for illustration. Row zero is in `Aj [0..1]` (column indices) and `Ax [0..1]` (values), starting at `Ap [0] = 0` and ending at `Ap [0+1]-1 = 1`. The list of column indices of row one is at `Aj [2..4]` and row two is in `Aj [5..6]`. The last row (three) appears `Aj [7..9]`, because `Ap [3] = 7` and `Ap [4]-1 = 10-1 = 9`. The corresponding numerical values appear in the same positions in `Ax`.

To iterate over the rows and entries of this matrix, the following code can be used (assuming it has type `GrB_FP64`):

```
int64_t nvals = Ap [nrows] ;
for (int64_t i = 0 ; i < nrows ; i++)
{
    // get A(i,:)
    for (int64_t p = Ap [i] ; p < Ap [i+1] ; p++)
    {
        // get A(i,j)
        int64_t  j = Aj [p] ;           // column index
        double aij = Ax [iso ? 0 : p] ;   // numerical value
    }
}
```

If successful, the three pointers `Ap`, `Aj`, and `Ax` are set to `NULL` on output. This denotes to the user application that it is no longer responsible for freeing these arrays. Internally, GraphBLAS has moved these arrays into its internal data structure. They will eventually

be freed no later than when the user does `GrB_free(&A)`, but they may be freed or resized later, if the matrix changes. If an unpack is performed, the freeing of these three arrays again becomes the responsibility of the user application.

The `GxB_Matrix_pack_CSR` function will rarely fail, since it allocates just $O(1)$ space. If it does fail, it returns `GrB_OUT_OF_MEMORY`, and it leaves the three user arrays unmodified. They are still owned by the user application, which is eventually responsible for freeing them with `free(Ap)`, etc.

6.11.8 GxB_Matrix_unpack_CSR: unpack a CSR matrix

```
GrB_Info GxB_Matrix_unpack_CSR  // unpack a CSR matrix
(
    GrB_Matrix A,       // matrix to unpack (type, nrows, ncols unchanged)
    GrB_Index **Ap,     // row "pointers"
    GrB_Index **Aj,     // column indices
    void **Ax,          // values
    GrB_Index *Ap_size, // size of Ap in bytes
    GrB_Index *Aj_size, // size of Aj in bytes
    GrB_Index *Ax_size, // size of Ax in bytes
    bool *iso,          // if true, A is iso
    bool *jumbled,      // if true, indices in each row may be unsorted
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_unpack_CSR` unpacks a matrix in CSR form.

If successful, the `GrB_Matrix A` is returned with no entries. The CSR format is in the three arrays `Ap`, `Aj`, and `Ax`. If the matrix has no entries, the `Aj` and `Ax` arrays may be returned as `NULL`; this is not an error, and `GxB_Matrix_pack_CSR` also allows these two arrays to be `NULL` on input when the matrix has no entries. After a successful unpack, the user application is responsible for freeing these three arrays via `free` (or the `free` function passed to `GxB_init`). The CSR format is described in Section 6.11.8.

If `jumbled` is returned as `false`, column indices will appear in ascending order within each row. If `jumbled` is returned as `true`, column indices may appear in any order within each row. If `jumbled` is passed in as `NULL`, then column indices will be returned in ascending order in each row. No duplicate column indices will appear in any row. `Ap [0]` is zero, and `Ap [nrows]` is equal to the number of entries in the matrix (`nvals`). The `Ap` array will be of size `nrows`+1 (or larger), and the `Aj` and `Ax` arrays will have size at least `nvals`.

This method takes $O(1)$ time if the matrix is already in CSR format internally. Otherwise, the matrix is converted to CSR format and then unpacked.

6.11.9 GxB_Matrix_pack_CSC: pack a CSC matrix

```
GrB_Info GxB_Matrix_pack_CSC      // pack a CSC matrix
(
    GrB_Matrix A,       // matrix to create (type, nrows, ncols unchanged)
    GrB_Index **Ap,     // col "pointers", Ap_size >= (ncols+1)*sizeof(int64_t)
    GrB_Index **Ai,     // row indices, Ai_size >= nvals(A)*sizeof(int64_t)
    void **Ax,          // values, Ax_size >= nvals(A) * (type size)
                        // or Ax_size >= (type size), if iso is true
    GrB_Index Ap_size,  // size of Ap in bytes
    GrB_Index Ai_size,  // size of Ai in bytes
    GrB_Index Ax_size,  // size of Ax in bytes
    bool iso,           // if true, A is iso
    bool jumbled,       // if true, indices in each column may be unsorted
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_pack_CSC` packs a matrix from 3 user arrays in CSC format. The `GrB_Matrix A` must exist on input with the right type and dimensions. No typecasting is done. The arguments are identical to `GxB_Matrix_pack_CSR`, except for how the 3 user arrays are interpreted. The column "pointer" array has size `ncols+1`. The row indices of the columns are in `Ai`, and if `jumbled` is false, they must appear in ascending order in each column. The corresponding numerical values are held in `Ax`. The row indices of column `j` are held in `Ai [Ap [j]...Ap [j+1]-1]`, and the corresponding numerical values are in the same locations in `Ax`.

The same matrix from Equation 1in the last section (repeated here):

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix} \tag{2}$$

is held in CSC form as follows:

```
int64_t Ap [ ] = { 0,             3,             6,        8,       10 } ;
int64_t Ai [ ] = { 0,   1,   3,   1,   2,   3,   0,   2,   1,   3   } ;
double  Ax [ ] = { 4.5, 3.1, 3.5, 2.9, 1.7, 0.4, 3.2, 3.0, 0.9, 1.0 } ;
```

That is, the row indices of column 1 (the second column) are in `Ai [3..5]`, and the values in the same place in `Ax`, since `Ap [1] = 3` and `Ap [2]-1 = 5`.

To iterate over the columns and entries of this matrix, the following code can be used (assuming it has type `GrB_FP64`):

```
int64_t nvals = Ap [ncols] ;
for (int64_t j = 0 ; j < ncols ; j++)
{
```

```
    // get A(:,j)
    for (int64_t p = Ap [j] ; p < Ap [j+1] ; p++)
    {
        // get A(i,j)
        int64_t  i = Ai [p] ;             // row index
        double aij = Ax [iso ? 0 : p] ;   // numerical value
    }
}
```

The method is identical to `GxB_Matrix_pack_CSR`; just the format is transposed.

If `Ap [ncols]` is zero, then the content of the `Ai` and `Ax` arrays is not accessed and they may be `NULL` on input (if not `NULL`, they are still freed and returned as `NULL`, if the method is successful).

6.11.10 GxB_Matrix_unpack_CSC: unpack a CSC matrix

```
GrB_Info GxB_Matrix_unpack_CSC  // unpack a CSC matrix
(
    GrB_Matrix A,       // matrix to unpack (type, nrows, ncols unchanged)
    GrB_Index **Ap,     // column "pointers"
    GrB_Index **Ai,     // row indices
    void **Ax,          // values
    GrB_Index *Ap_size, // size of Ap in bytes
    GrB_Index *Ai_size, // size of Ai in bytes
    GrB_Index *Ax_size, // size of Ax in bytes
    bool *iso,          // if true, A is iso
    bool *jumbled,      // if true, indices in each column may be unsorted
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_unpack_CSC` unpacks a matrix in CSC form.

If successful, the `GrB_Matrix A` is returned with no entries. The CSC format is in the three arrays `Ap`, `Ai`, and `Ax`. If the matrix has no entries, `Ai` and `Ax` arrays are returned as `NULL`; this is not an error, and `GxB_Matrix_pack_CSC` also allows these two arrays to be `NULL` on input when the matrix has no entries. After a successful unpack, the user application is responsible for freeing these three arrays via `free` (or the `free` function passed to `GxB_init`). The CSC format is described in Section 6.11.10.

This method takes $O(1)$ time if the matrix is already in CSC format internally. Otherwise, the matrix is converted to CSC format and then unpacked.

6.11.11 GxB_Matrix_pack_HyperCSR: pack a HyperCSR matrix

```
GrB_Info GxB_Matrix_pack_HyperCSR      // pack a hypersparse CSR matrix
(
    GrB_Matrix A,       // matrix to create (type, nrows, ncols unchanged)
    GrB_Index **Ap,     // row "pointers", Ap_size >= (nvec+1)*sizeof(int64_t)
    GrB_Index **Ah,     // row indices, Ah_size >= nvec*sizeof(int64_t)
    GrB_Index **Aj,     // column indices, Aj_size >= nvals(A)*sizeof(int64_t)
    void **Ax,          // values, Ax_size >= nvals(A) * (type size)
                        // or Ax_size >= (type size), if iso is true
    GrB_Index Ap_size,  // size of Ap in bytes
    GrB_Index Ah_size,  // size of Ah in bytes
    GrB_Index Aj_size,  // size of Aj in bytes
    GrB_Index Ax_size,  // size of Ax in bytes
    bool iso,           // if true, A is iso
    GrB_Index nvec,     // number of rows that appear in Ah
    bool jumbled,       // if true, indices in each row may be unsorted
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_pack_HyperCSR` packs a matrix in hypersparse CSR format. The hypersparse HyperCSR format is identical to the CSR format, except that the `Ap` array itself becomes sparse, if the matrix has rows that are completely empty. An array `Ah` of size `nvec` provides a list of rows that appear in the data structure. For example, consider Equation 3, which is a sparser version of the matrix in Equation 1. Row 2 and column 1 of this matrix are all zero.

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 0 & 0 & 0.9 \\ 0 & 0 & 0 & 0 \\ 3.5 & 0 & 0 & 1.0 \end{bmatrix} \tag{3}$$

The conventional CSR format would appear as follows. Since the third row (row 2) is all zero, accessing `Ai [Ap [2] ... Ap [3]-1]` gives an empty set (`[2..1]`), and the number of entries in this row is `Ap [i+1] - Ap [i] = Ap [3] - Ap [2] = 0`.

```
int64_t Ap [ ] = { 0,        2,2,      4,       5 } ;
int64_t Aj [ ] = { 0,   2,   0,   3,   0    3   }
double  Ax [ ] = { 4.5, 3.2, 3.1, 0.9, 3.5, 1.0 } ;
```

A hypersparse CSR format for this same matrix would discard these duplicate integers in `Ap`. Doing so requires another array, `Ah`, that keeps track of the rows that appear in the data structure.

```
int64_t nvec = 3 ;
int64_t Ah [ ] = { 0,        1,        3        } ;
```

```
int64_t Ap [ ] = { 0,        2,        4,       5 } ;
int64_t Aj [ ] = { 0,   2,   0,   3,   0    3   }
double  Ax [ ] = { 4.5, 3.2, 3.1, 0.9, 3.5, 1.0 } ;
```

Note that the `Aj` and `Ax` arrays are the same in the CSR and HyperCSR formats. If `jumbled` is false, the row indices in `Ah` must appear in ascending order, and no duplicates can appear. To iterate over this data structure (assuming it has type `GrB_FP64`):

```
int64_t nvals = Ap [nvec] ;
for (int64_t k = 0 ; k < nvec ; k++)
{
    int64_t i = Ah [k] ;                // row index
    // get A(i,:)
    for (int64_t p = Ap [k] ; p < Ap [k+1] ; p++)
    {
        // get A(i,j)
        int64_t  j = Aj [p] ;             // column index
        double aij = Ax [iso ? 0 : p] ;   // numerical value
    }
}
```

This is more complex than the CSR format, but it requires at most $O(e)$ space, where A is m-by-n with $e =$ `nvals` entries. The CSR format requires $O(m+e)$ space. If $e << m$, then the size $m+1$ of `Ap` can dominate the memory required. In the hypersparse form, `Ap` takes on size `nvec+1`, and `Ah` has size `nvec`, where `nvec` is the number of rows that appear in the data structure. The CSR format can be viewed as a dense array (of size `nrows`) of sparse row vectors. By contrast, the hypersparse CSR format is a sparse array (of size `nvec`) of sparse row vectors.

The pack takes $O(1)$ time. If successful, the four arrays `Ah`, `Ap`, `Aj`, and `Ax` are returned as `NULL`, and the hypersparse `GrB_Matrix A` is modified to contain the entries they describe.

If the matrix has no entries, then the content of the `Aj` and `Ax` arrays is not accessed and they may be `NULL` on input (if not `NULL`, they are still freed and returned as `NULL`, if the method is successful).

6.11.12 GxB_Matrix_unpack_HyperCSR: unpack a HyperCSR matrix

```
GrB_Info GxB_Matrix_unpack_HyperCSR  // unpack a hypersparse CSR matrix
(
    GrB_Matrix A,       // matrix to unpack (type, nrows, ncols unchanged)
    GrB_Index **Ap,     // row "pointers"
    GrB_Index **Ah,     // row indices
    GrB_Index **Aj,     // column indices
    void **Ax,          // values
    GrB_Index *Ap_size, // size of Ap in bytes
    GrB_Index *Ah_size, // size of Ah in bytes
    GrB_Index *Aj_size, // size of Aj in bytes
    GrB_Index *Ax_size, // size of Ax in bytes
    bool *iso,          // if true, A is iso
    GrB_Index *nvec,    // number of rows that appear in Ah
    bool *jumbled,      // if true, indices in each row may be unsorted
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_unpack_HyperCSR` unpacks a matrix in HyperCSR format.

If successful, the `GrB_Matrix A` is returned with no entries. The number of non-empty rows is `nvec`. The hypersparse CSR format is in the four arrays `Ah`, `Ap`, `Aj`, and `Ax`. If the matrix has no entries, the `Aj` and `Ax` arrays are returned as `NULL`; this is not an error. After a successful unpack, the user application is responsible for freeing these three arrays via `free` (or the `free` function passed to `GxB_init`). The hypersparse CSR format is described in Section 6.11.11.

This method takes $O(1)$ time if the matrix is already in HyperCSR format internally. Otherwise, the matrix is converted to HyperCSR format and then unpacked.

6.11.13 GxB_Matrix_pack_HyperCSC: pack a HyperCSC matrix

```
GrB_Info GxB_Matrix_pack_HyperCSR      // pack a hypersparse CSR matrix
(
    GrB_Matrix A,       // matrix to create (type, nrows, ncols unchanged)
    GrB_Index **Ap,     // row "pointers", Ap_size >= (nvec+1)*sizeof(int64_t)
    GrB_Index **Ah,     // row indices, Ah_size >= nvec*sizeof(int64_t)
    GrB_Index **Aj,     // column indices, Aj_size >= nvals(A)*sizeof(int64_t)
    void **Ax,          // values, Ax_size >= nvals(A) * (type size)
                        // or Ax_size >= (type size), if iso is true
    GrB_Index Ap_size,  // size of Ap in bytes
    GrB_Index Ah_size,  // size of Ah in bytes
    GrB_Index Aj_size,  // size of Aj in bytes
    GrB_Index Ax_size,  // size of Ax in bytes
    bool iso,           // if true, A is iso
    GrB_Index nvec,     // number of rows that appear in Ah
    bool jumbled,       // if true, indices in each row may be unsorted
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_pack_HyperCSC` packs a matrix in hypersparse CSC format. It is identical to `GxB_Matrix_pack_HyperCSR`, except the data structure defined by the four arrays `Ah`, `Ap`, `Ai`, and `Ax` holds the matrix as a sparse array of `nvec` sparse column vectors. The following code iterates over the matrix, assuming it has type `GrB_FP64`:

```
int64_t nvals = Ap [nvec] ;
for (int64_t k = 0 ; k < nvec ; k++)
{
    int64_t j = Ah [k] ;                // column index
    // get A(:,j)
    for (int64_t p = Ap [k] ; p < Ap [k+1] ; p++)
    {
        // get A(i,j)
        int64_t  i = Ai [p] ;             // row index
        double aij = Ax [iso ? 0 : p] ;   // numerical value
    }
}
```

6.11.14 GxB_Matrix_unpack_HyperCSC: unpack a HyperCSC matrix

```
GrB_Info GxB_Matrix_unpack_HyperCSR  // unpack a hypersparse CSR matrix
(
    GrB_Matrix A,       // matrix to unpack (type, nrows, ncols unchanged)
    GrB_Index **Ap,     // row "pointers"
    GrB_Index **Ah,     // row indices
    GrB_Index **Aj,     // column indices
    void **Ax,          // values
    GrB_Index *Ap_size, // size of Ap in bytes
    GrB_Index *Ah_size, // size of Ah in bytes
    GrB_Index *Aj_size, // size of Aj in bytes
    GrB_Index *Ax_size, // size of Ax in bytes
    bool *iso,          // if true, A is iso
    GrB_Index *nvec,    // number of rows that appear in Ah
    bool *jumbled,      // if true, indices in each row may be unsorted
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_unpack_HyperCSC` unpacks a matrix in HyperCSC form.

If successful, the `GrB_Matrix A` is returned with no entries. The number of non-empty rows is in `nvec`. The hypersparse CSC format is in the four arrays `Ah`, `Ap`, `Ai`, and `Ax`. If the matrix has no entries, the `Ai` and `Ax` arrays are returned as `NULL`; this is not an error. After a successful unpack, the user application is responsible for freeing these three arrays via `free` (or the `free` function passed to `GxB_init`). The hypersparse CSC format is described in Section 6.11.13.

This method takes $O(1)$ time if the matrix is already in HyperCSC format internally. Otherwise, the matrix is converted to HyperCSC format and then unpacked.

6.11.15 GxB_Matrix_pack_BitmapR: pack a BitmapR matrix

```
GrB_Info GxB_Matrix_pack_BitmapR  // pack a bitmap matrix, held by row
(
    GrB_Matrix A,       // matrix to create (type, nrows, ncols unchanged)
    int8_t **Ab,        // bitmap, Ab_size >= nrows*ncols
    void **Ax,          // values, Ax_size >= nrows*ncols * (type size)
                        // or Ax_size >= (type size), if iso is true
    GrB_Index Ab_size,  // size of Ab in bytes
    GrB_Index Ax_size,  // size of Ax in bytes
    bool iso,           // if true, A is iso
    GrB_Index nvals,    // # of entries in bitmap
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_pack_BitmapR` packs a matrix from 2 user arrays in BitmapR format. The matrix must exist on input with the right type and dimensions. No typecasting is done.

The `GrB_Matrix A` is populated from the arrays `Ab` and `Ax`, each of which are size `nrows*ncols`. Both arrays must have been created with the ANSI C `malloc`, `calloc`, or `realloc` functions (by default), or by the corresponding `malloc_function`, `calloc_function`, or `realloc_function` provided to `GxB_init`. These arrays define the pattern and values of the new matrix `A`:

- `int8_t Ab [nrows*ncols] ;` The `Ab` array defines which entries of `A` are present. If `Ab[i*ncols+j]=1`, then the entry $A(i,j)$ is present, with value `Ax[i*ncols+j]`. If `Ab[i*ncols+j]=0`, then the entry $A(i,j)$ is not present. The `Ab` array must contain only 0s and 1s. The `nvals` input must exactly match the number of 1s in the `Ab` array.
- `ctype Ax [nrows*ncols] ;` The `Ax` array defines the values of entries in the matrix. It is passed in as a `(void *)` pointer, but it must point to an array of size `nrows*ncols` values, each of size `sizeof(ctype)`, where `ctype` is the exact type in C that corresponds to the `GrB_Type type` parameter. That is, if `type` is `GrB_INT32`, then `ctype` is `int32_t`. User types may be used, just the same as built-in types. If `Ab[p]` is zero, the value of `Ax[p]` is ignored.

To iterate over the rows and entries of this matrix, the following code can be used (assuming it has type `GrB_FP64`):

```
for (int64_t i = 0 ; i < nrows ; i++)
{
    // get A(i,:)
    for (int64_t j = 0 ; j < ncols ; j++)
    {
        // get A(i,j)
        int64_t p = i*ncols + j ;
```

```
            if (Ab [p])
            {
                double aij = Ax [iso ? 0 : p] ;   // numerical value
            }
        }
    }
```

On successful pack of `A`, the two pointers `Ab`, `Ax`, are set to `NULL` on output. This denotes to the user application that it is no longer responsible for freeing these arrays. Internally, GraphBLAS has moved these arrays into its internal data structure. They will eventually be freed no later than when the user does `GrB_free(&A)`, but they may be freed or resized later, if the matrix changes. If an unpack is performed, the freeing of these three arrays again becomes the responsibility of the user application.

The `GxB_Matrix_pack_BitmapR` function will rarely fail, since it allocates just $O(1)$ space. If it does fail, it returns `GrB_OUT_OF_MEMORY`, and it leaves the two user arrays unmodified. They are still owned by the user application, which is eventually responsible for freeing them with `free(Ab)`, etc.

6.11.16 GxB_Matrix_unpack_BitmapR: unpack a BitmapR matrix

```
GrB_Info GxB_Matrix_unpack_BitmapR  // unpack a bitmap matrix, by row
(
    GrB_Matrix A,       // matrix to unpack (type, nrows, ncols unchanged)
    int8_t **Ab,        // bitmap
    void **Ax,          // values
    GrB_Index *Ab_size, // size of Ab in bytes
    GrB_Index *Ax_size, // size of Ax in bytes
    bool *iso,          // if true, A is iso
    GrB_Index *nvals,   // # of entries in bitmap
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_unpack_BitmapR` unpacks a matrix in BitmapR form. If successful, the `GrB_Matrix A` is returned with no entries. The number of entries is in `nvals`. The BitmapR format is two arrays `Ab`, and `Ax`. After an unpack, the user application is responsible for freeing these arrays via `free` (or the `free` function passed to `GxB_init`). The BitmapR format is described in Section 6.11.15. If `Ab[p]` is zero, the value of `Ax[p]` is undefined. This method takes $O(1)$ time if the matrix is already in BitmapR format.

6.11.17 GxB_Matrix_pack_BitmapC: pack a BitmapC matrix

```
GrB_Info GxB_Matrix_pack_BitmapC  // pack a bitmap matrix, held by column
(
    GrB_Matrix A,       // matrix to create (type, nrows, ncols unchanged)
    int8_t **Ab,        // bitmap, Ab_size >= nrows*ncols
    void **Ax,          // values, Ax_size >= nrows*ncols * (type size)
                        // or Ax_size >= (type size), if iso is true
    GrB_Index Ab_size,  // size of Ab in bytes
    GrB_Index Ax_size,  // size of Ax in bytes
    bool iso,           // if true, A is iso
    GrB_Index nvals,    // # of entries in bitmap
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_pack_BitmapC` packs a matrix from 2 user arrays in BitmapC format. It is identical to `GxB_Matrix_pack_BitmapR`, except that the entry $A(i,j)$ is held in `Ab[i+j*nrows]` and `Ax[i+j*nrows]`, in column-major format.

6.11.18 GxB_Matrix_unpack_BitmapC: unpack a BitmapC matrix

```
GrB_Info GxB_Matrix_unpack_BitmapC  // unpack a bitmap matrix, by col
(
    GrB_Matrix A,       // matrix to unpack (type, nrows, ncols unchanged)
    int8_t **Ab,        // bitmap
    void **Ax,          // values
    GrB_Index *Ab_size, // size of Ab in bytes
    GrB_Index *Ax_size, // size of Ax in bytes
    bool *iso,          // if true, A is iso
    GrB_Index *nvals,   // # of entries in bitmap
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_unpack_BitmapC` unpacks a matrix in BitmapC form. It is identical to `GxB_Matrix_unpack_BitmapR`, except that the entry $A(i,j)$ is held in `Ab[i+j*nrows]` and `Ax[i+j*nrows]`, in column-major format.

6.11.19 GxB_Matrix_pack_FullR: pack a FullR matrix

```
GrB_Info GxB_Matrix_pack_FullR  // pack a full matrix, held by row
(
    GrB_Matrix A,       // matrix to create (type, nrows, ncols unchanged)
    void **Ax,          // values, Ax_size >= nrows*ncols * (type size)
                        // or Ax_size >= (type size), if iso is true
    GrB_Index Ax_size,  // size of Ax in bytes
    bool iso,           // if true, A is iso
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_pack_FullR` packs a matrix from a user array in FullR format. For the `FullR` format, t value of $A(i,j)$ is `Ax[i*ncols+j]`. To iterate over the rows and entries of this matrix, the following code can be used (assuming it has type `GrB_FP64`). If `A` is both full and iso, it takes $O(1)$ memory, regardless of `nrows` and `ncols`.

```
for (int64_t i = 0 ; i < nrows ; i++)
{
    for (int64_t j = 0 ; j < ncols ; j++)
    {
        int64_t p = i*ncols + j ;
        double aij = Ax [iso ? 0 : p] ;   // numerical value of A(i,j)
    }
}
```

6.11.20 GxB_Matrix_unpack_FullR: unpack a FullR matrix

```
GrB_Info GxB_Matrix_unpack_FullR  // unpack a full matrix, by row
(
    GrB_Matrix A,       // matrix to unpack (type, nrows, ncols unchanged)
    void **Ax,          // values
    GrB_Index *Ax_size, // size of Ax in bytes
    bool *iso,          // if true, A is iso
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_unpack_FullR` unpacks a matrix in FullR form. It is identical to `GxB_Matrix_unpack_BitmapR`, except that all entries must be present. Prior to unpack, `GrB_Matrix_nvals (&nvals, A)` must return `nvals` equal to `nrows*ncols`. Otherwise, if the `A` is unpacked with `GxB_Matrix_unpack_FullR`, an error is returned (`GrB_INVALID_VALUE`) and the matrix is not unpacked.

6.11.21 GxB_Matrix_pack_FullC: pack a FullC matrix

```
GrB_Info GxB_Matrix_pack_FullC  // pack a full matrix, held by column
(
    GrB_Matrix A,       // matrix to create (type, nrows, ncols unchanged)
    void **Ax,          // values, Ax_size >= nrows*ncols * (type size)
                        // or Ax_size >= (type size), if iso is true
    GrB_Index Ax_size,  // size of Ax in bytes
    bool iso,           // if true, A is iso
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_pack_FullC` packs a matrix from a user arrays in FullC format. For the `FullC` format, the value of $A(i,j)$ is `Ax[i+j*nrows]`. To iterate over the rows and entries of this matrix, the following code can be used (assuming it has type `GrB_FP64`). If `A` is both full and iso, it takes $O(1)$ memory, regardless of `nrows` and `ncols`.

```
for (int64_t i = 0 ; i < nrows ; i++)
{
    for (int64_t j = 0 ; j < ncols ; j++)
    {
        int64_t p = i + j*nrows ;
        double aij = Ax [iso ? 0 : p] ;   // numerical value of A(i,j)
    }
}
```

6.11.22 GxB_Matrix_unpack_FullC: unpack a FullC matrix

```
GrB_Info GxB_Matrix_unpack_FullC  // unpack a full matrix, by column
(
    GrB_Matrix A,       // matrix to unpack (type, nrows, ncols unchanged)
    void **Ax,          // values
    GrB_Index *Ax_size, // size of Ax in bytes
    bool *iso,          // if true, A is iso
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_unpack_FullC` unpacks a matrix in FullC form. It is identical to `GxB_Matrix_unpack_BitmapC`, except that all entries must be present. That is, prior to unpack, `GrB_Matrix_nvals (&nvals, A)` must return `nvals` equal to `nrows*ncols`. Otherwise, if the `A` is unpacked with `GxB_Matrix_unpack_FullC`, an error is returned (`GrB_INVALID_VALUE`) and the matrix is not unpacked.

6.12 GraphBLAS import/export: using copy semantics

The v2.0 C API includes import/export methods for matrices (not vectors) using a different strategy as compared to the `GxB*pack/unpack*` methods. The `GxB` methods are based on *move semantics*, in which ownership of arrays is passed between SuiteSparse:GraphBLAS and the user application. This allows the `GxB*pack/unpack*` methods to work in $O(1)$ time, and require no additional memory, but it requires that GraphBLAS and the user application agree on which memory manager to use. This is done via `GxB_init`. This allows GraphBLAS to `malloc` an array that can be later `free`d by the user application, and visa versa.

The `GrB` import/export methods take a different approach. The data is always copied in and out between the opaque GraphBLAS matrix and the user arrays. This takes $\Omega(e)$ time, if the matrix has e entries, and requires more memory. It has the advantage that it does not require GraphBLAS and the user application to agree on what memory manager to use, since no ownership of allocated arrays is changed.

The format for `GrB_Matrix_import` and `GrB_Matrix_export` is controlled by the following enum:

```
typedef enum
{
    GrB_CSR_FORMAT = 0,     // CSR format (equiv to GxB_SPARSE with GxB_BY_ROW)
    GrB_CSC_FORMAT = 1,     // CSC format (equiv to GxB_SPARSE with GxB_BY_COL)
    GrB_COO_FORMAT = 2      // triplet format (like input to GrB*build)
}
GrB_Format ;
```

6.12.1 GrB_Matrix_import: **import a matrix**

```
GrB_Info GrB_Matrix_import  // import a matrix
(
    GrB_Matrix *A,          // handle of matrix to create
    GrB_Type type,          // type of matrix to create
    GrB_Index nrows,        // number of rows of the matrix
    GrB_Index ncols,        // number of columns of the matrix
    const GrB_Index *Ap,    // pointers for CSR, CSC, column indices for COO
    const GrB_Index *Ai,    // row indices for CSR, CSC
    const <type> *Ax,       // values
    GrB_Index Ap_len,       // number of entries in Ap (not # of bytes)
    GrB_Index Ai_len,       // number of entries in Ai (not # of bytes)
    GrB_Index Ax_len,       // number of entries in Ax (not # of bytes)
    GrB_Format format       // import format
) ;
```

The `GrB_Matrix_import` method copies from user-provided arrays into an opaque `GrB_Matrix` and `GrB_Matrix_export` copies data out, from an opaque `GrB_Matrix` into user-provided arrays.

The suffix `TYPE` in the prototype above is one of `BOOL`, `INT8`, `INT16`, etc, for built-n types, or `UDT` for user-defined types. The type of the `Ax` array must match this type. No typecasting is performed.

Unlike the `GxB` pack/unpack methods, memory is not handed off between the user application and GraphBLAS. The three arrays `Ap`, `Ai`. and `Ax` are not modified, and are still owned by the user application when the method finishes.

The matrix can be imported in one of three different formats:

- `GrB_CSR_FORMAT`: Compressed-row format. `Ap` is an array of size `nrows+1`. The arrays `Ai` and `Ax` are of size `nvals = Ap [nrows]`, and `Ap[0]` must be zero. The column indices of entries in the `i`th row appear in `Ai[Ap[i]...Ap[i+1]-1]`, and the values of those entries appear in the same locations in `Ax`. The column indices need not be in any particular order.
- `GrB_CSC_FORMAT`: Compressed-column format. `Ap` is an array of size `ncols+1`. The arrays `Ai` and `Ax` are of size `nvals = Ap [ncols]`, and `Ap[0]` must be zero. The row indices of entries in the `j`th column appear in `Ai[Ap[j]...Ap[j+1]-1]`, and the values of those entries appear in the same locations in `Ax`. The row indices need not be in any particular order.
- `GrB_COO_FORMAT`: Coordinate format. This is the same format as `GrB_Matrix_build`. The three arrays `Ap`, `Ai`, and `Ax` have the same size. The `k`th tuple has row index `Ai[k]`, column index `Ap[k]`, and value `Ax[k]`. The tuples can appear any order, but no duplicates are permitted.

6.12.2 GrB_Matrix_export: **export a matrix**

```
GrB_Info GrB_Matrix_export  // export a matrix
(
    GrB_Index *Ap,          // pointers for CSR, CSC, column indices for COO
    GrB_Index *Ai,          // col indices for CSR/COO, row indices for CSC
    <type> *Ax,             // values (must match the type of A_input)
    GrB_Index *Ap_len,      // number of entries in Ap (not # of bytes)
    GrB_Index *Ai_len,      // number of entries in Ai (not # of bytes)
    GrB_Index *Ax_len,      // number of entries in Ax (not # of bytes)
    GrB_Format format,      // export format
    GrB_Matrix A            // matrix to export
) ;
```

`GrB_Matrix_export` copies the contents of a matrix into three user-provided arrays, using any one of the three different formats described in Section 6.12.1. The size of the arrays must be at least as large as the lengths returned by `GrB_Matrix_exportSize`. The matrix `A` is not modified.

On input, the size of the three arrays `Ap`, `Ai`, and `Ax` is given by `Ap_len`, `Ai_len`, and `Ax_len`, respectively. These values are in terms of the number of entries in these arrays, not the number of bytes. On output, these three value are adjusted to report the number of entries written to the three arrays.

The suffix `TYPE` in the prototype above is one of `BOOL`, `INT8`, `INT16`, etc, for built-n types, or `UDT` for user-defined types. The type of the `Ax` array must match this type. No typecasting is performed.

6.12.3 GrB_Matrix_exportSize: determine size of export

```
GrB_Info GrB_Matrix_exportSize  // determine sizes of user arrays for export
(
    GrB_Index *Ap_len,      // # of entries required for Ap (not # of bytes)
    GrB_Index *Ai_len,      // # of entries required for Ai (not # of bytes)
    GrB_Index *Ax_len,      // # of entries required for Ax (not # of bytes)
    GrB_Format format,      // export format
    GrB_Matrix A            // matrix to export
) ;
```

Returns the required sizes of the arrays `Ap`, `Ai`, and `Ax` for exporting a matrix using `GrB_Matrix_export`, using the same `format`.

6.12.4 GrB_Matrix_exportHint: determine best export format

```
GrB_Info GrB_Matrix_exportHint  // suggest the best export format
(
    GrB_Format *format,     // export format
    GrB_Matrix A            // matrix to export
) ;
```

This method suggests the most efficient format for the export of a given matrix. For SuiteSparse:GraphBLAS, the hint depends on the current format of the `GrB_Matrix`:

- `GxB_SPARSE`, `GxB_BY_ROW`: export as `GrB_CSR_FORMAT`
- `GxB_SPARSE`, `GxB_BY_COL`: export as `GrB_CSC_FORMAT`
- `GxB_HYPERSPARSE`: export as `GrB_COO_FORMAT`
- `GxB_BITMAP`, `GxB_BY_ROW`: export as `GrB_CSR_FORMAT`
- `GxB_BITMAP`, `GxB_BY_COL`: export as `GrB_CSC_FORMAT`
- `GxB_FULL`, `GxB_BY_ROW`: export as `GrB_CSR_FORMAT`
- `GxB_FULL`, `GxB_BY_COL`: export as `GrB_CSC_FORMAT`

6.13 Sorting methods

GxB_Matrix_sort provides a mechanism to sort all the rows or all the columns of a matrix, and GxB_Vector_sort sorts all the entries in a vector.

6.13.1 GxB_Vector_sort: sort a vector

```
GrB_Info GxB_sort
(
    // output:
    GrB_Vector w,           // vector of sorted values
    GrB_Vector p,           // vector containing the permutation
    // input
    GrB_BinaryOp op,        // comparator op
    GrB_Vector u,           // vector to sort
    const GrB_Descriptor desc
) ;
```

GxB_Vector_sort is identical to sorting the single column of an n-by-1 matrix. The descriptor is ignored, except to control the number of threads to use. Refer to Section 6.13.2 for details.

6.13.2 GxB_Matrix_sort: sort the rows/columns of a matrix

```
GrB_Info GxB_sort
(
    // output:
    GrB_Matrix C,           // matrix of sorted values
    GrB_Matrix P,           // matrix containing the permutations
    // input
    GrB_BinaryOp op,        // comparator op
    GrB_Matrix A,           // matrix to sort
    const GrB_Descriptor desc
) ;
```

GxB_Matrix_sort sorts all the rows or all the columns of a matrix. Each row (or column) is sorted separately. The rows are sorted by default. To sort the columns, use GrB_DESC_T0. A comparator operator is provided to define the sorting order (ascending or descending). For example, to sort a GrB_FP64 matrix in ascending order, use GrB_LT_FP64 as the op, and to sort in descending order, use GrB_GT_FP64.

The op must have a return value of GrB_BOOL, and the types of its two inputs must be the same. The entries in A are typecasted to the inputs of the op, if necessary. Matrices with user-defined types can be sorted with a user-defined comparator operator, whose two input types must match the type of A, and whose output is GrB_BOOL.

The two matrix outputs are `C` and `P`. Any entries present on input in `C` or `P` are discarded on output. The type of `C` must match the type of `A` exactly. The dimensions of `C`, `P`, and `A` must also match exactly (even with the `GrB_DESC_T0` descriptor).

With the default sort (by row), suppose `A(i,:)` contains `k` entries. In this case, `C(i,0:k-1)` contains the values of those entries in sorted order, and `P(i,0:k-1)` contains their corresponding column indices in the matrix `A`. If two values are the same, ties are broken according column index.

If the matrix is sorted by column, and `A(:,j)` contains `k` entries, then `C(0:k-1,j)` contains the values of those entries in sorted order, and `P(0:k-1,j)` contains their corresponding row indices in the matrix `A`. If two values are the same, ties are broken according row index.

The outputs `C` and `P` are both optional; either one (but not both) may be `NULL`, in which case that particular output matrix is not computed.

6.14 GraphBLAS descriptors: GrB_Descriptor

A GraphBLAS *descriptor* modifies the behavior of a GraphBLAS operation. If the descriptor is `GrB_NULL`, defaults are used.

The access to these parameters and their values is governed by two `enum` types, `GrB_Desc_Field` and `GrB_Desc_Value`:

```
#define GxB_NTHREADS 5  // for both GrB_Desc_field and GxB_Option_field
#define GxB_CHUNK 7
typedef enum
{
    GrB_OUTP = 0,   // descriptor for output of a method
    GrB_MASK = 1,   // descriptor for the mask input of a method
    GrB_INP0 = 2,   // descriptor for the first input of a method
    GrB_INP1 = 3,   // descriptor for the second input of a method
    GxB_DESCRIPTOR_NTHREADS = GxB_NTHREADS,   // number of threads to use
    GxB_DESCRIPTOR_CHUNK = GxB_CHUNK,   // chunk size for small problems
    GxB_AxB_METHOD = 1000, // descriptor for selecting C=A*B algorithm
    GxB_SORT = 35   // control sort in GrB_mxm
    GxB_COMPRESSION = 36,   // select compression for serialize
    GxB_IMPORT = 37,        // secure vs fast pack
}
GrB_Desc_Field ;

typedef enum
{
    // for all GrB_Descriptor fields:
    GxB_DEFAULT = 0,    // default behavior of the method
    // for GrB_OUTP only:
    GrB_REPLACE = 1,    // clear the output before assigning new values to it
    // for GrB_MASK only:
    GrB_COMP = 2,       // use the complement of the mask
    GrB_STRUCTURE = 4,  // use the structure of the mask
    // for GrB_INP0 and GrB_INP1 only:
    GrB_TRAN = 3,       // use the transpose of the input
    // for GxB_AxB_METHOD only:
    GxB_AxB_GUSTAVSON = 1001,   // gather-scatter saxpy method
    GxB_AxB_DOT       = 1003,   // dot product
    GxB_AxB_HASH      = 1004,   // hash-based saxpy method
    GxB_AxB_SAXPY     = 1005    // saxpy method (any kind)
    // for GxB_IMPORT only:
    GxB_SECURE_IMPORT = 502     // GxB*_pack* methods trust their input data
}
GrB_Desc_Value ;
```

- `GrB_OUTP` is a parameter that modifies the output of a GraphBLAS operation. In the default case, the output is not cleared, and $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ then $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ are computed as-is, where $\mathbf{T}$ is the results of the particular GraphBLAS operation.

  In the non-default case, $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ is first computed, using the results of $\mathbf{T}$ and the accumulator $\odot$. After this is done, if the `GrB_OUTP` descriptor field is set to `GrB_REPLACE`, then the output is cleared of its entries. Next, the assignment $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ is performed.

- `GrB_MASK` is a parameter that modifies the `Mask`, even if the mask is not present.

  If this parameter is set to its default value, and if the mask is not present (`Mask==NULL`) then implicitly `Mask(i,j)=1` for all `i` and `j`. If the mask is present then `Mask(i,j)=1` means that `C(i,j)` is to be modified by the $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ update. Otherwise, if `Mask(i,j)=0`, then `C(i,j)` is not modified, even if `Z(i,j)` is an entry with a different value; that value is simply discarded.

  If the `GrB_MASK` parameter is set to `GrB_COMP`, then the use of the mask is complemented. In this case, if the mask is not present (`Mask==NULL`) then implicitly `Mask(i,j)=0` for all `i` and `j`. This means that none of $\mathbf{C}$ is modified and the entire computation of $\mathbf{Z}$ might as well have been skipped. That is, a complemented empty mask means no modifications are made to the output object at all, except perhaps to clear it in accordance with the `GrB_OUTP` descriptor. With a complemented mask, if the mask is present then `Mask(i,j)=0` means that `C(i,j)` is to be modified by the $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ update. Otherwise, if `Mask(i,j)=1`, then `C(i,j)` is not modified, even if `Z(i,j)` is an entry with a different value; that value is simply discarded.

  If the `GrB_MASK` parameter is set to `GrB_STRUCTURE`, then the values of the mask are ignored, and just the pattern of the entries is used. Any entry `M(i,j)` in the pattern is treated as if it were true.

  The `GrB_COMP` and `GrB_STRUCTURE` settings can be combined, either by setting the mask option twice (once with each value), or by setting the mask option to `GrB_COMP+GrB_STRUCTURE` (the latter is an extension to the specification).

  Using a parameter to complement the `Mask` is very useful because constructing the actual complement of a very sparse mask is impossible since it has too many entries. If the number of places in `C` that should be modified is very small, then use a sparse mask without complementing it. If the number of places in `C` that should be protected from modification is very small, then use a sparse mask to indicate those places, and use a descriptor `GrB_MASK` that complements the use of the mask.

- `GrB_INP0` and `GrB_INP1` modify the use of the first and second input matrices `A` and `B` of the GraphBLAS operation.

  If the `GrB_INP0` is set to `GrB_TRAN`, then `A` is transposed before using it in the operation. Likewise, if `GrB_INP1` is set to `GrB_TRAN`, then the second input, typically called `B`, is transposed.

  Vectors and scalars are never transposed via the descriptor. If a method's first parameter is a matrix and the second a vector or scalar, then `GrB_INP0` modifies the matrix parameter and `GrB_INP1` is ignored. If a method's first parameter is

a vector or scalar and the second a matrix, then `GrB_INP1` modifies the matrix parameter and `GrB_INP0` is ignored.

To clarify this in each function, the inputs are labeled as `first input:` and `second input:` in the function signatures.

- `GxB_AxB_METHOD` suggests the method that should be used to compute `C=A*B`. All the methods compute the same result, except they may have different floating-point roundoff errors. This descriptor should be considered as a hint; SuiteSparse:GraphBLAS is free to ignore it.

  - `GxB_DEFAULT` means that a method is selected automatically.
  - `GxB_AxB_SAXPY`: select any saxpy-based method: `GxB_AxB_GUSTAVSON`, and/or `GxB_AxB_HASH`, or any mix of the two, in contrast to the dot-product method.
  - `GxB_AxB_GUSTAVSON`: an extended version of Gustavson's method [Gus78], which is a very good general-purpose method, but sometimes the workspace can be too large. Assuming all matrices are stored by column, it computes `C(:,j)=A*B(:,j)` with a sequence of *saxpy* operations (`C(:,j)+=A(:,k)*B(k:,j)` for each nonzero `B(k,j)`). In the *coarse Gustavson* method, each internal thread requires workspace of size m, to the number of rows of `C`, which is not suitable if the matrices are extremely sparse or if there are many threads. For the *fine Gustavson* method, threads can share workspace and update it via atomic operations. If all matrices are stored by row, then it computes `C(i,:)=A(i,:)*B` in a sequence of sparse *saxpy* operations, and using workspace of size n per thread, or group of threads, corresponding to the number of columns of `C`.
  - `GxB_AxB_HASH`: a hash-based method, based on [NMAB18]. It is very efficient for hypersparse matrices, matrix-vector-multiply, and when $|\mathbf{B}|$ is small. SuiteSparse:GraphBLAS includes a *coarse hash* method, in which each thread has its own hash workspace, and a *fine hash* method, in which groups of threads share a single hash workspace, as concurrent data structure, using atomics.
  - `GxB_AxB_DOT`: computes `C(i,j)=A(i,:)*B(j,:)'`, for each entry `C(i,j)`. If the mask is present and not complemented, only entries for which `M(i,j)=1` are computed. This is a very specialized method that works well only if the mask is present, very sparse, and not complemented, when `C` is small, or when `C` is bitmap or full. For example, it works very well when `A` and `B` are tall and thin, and `C<M>=A*B'` or `C=A*B'` are computed. These expressions assume all matrices are in CSR format. If in CSC format, then the dot-product method used for `A'*B`. The method is impossibly slow if `C` is large and the mask is not present, since it takes $\Omega(mn)$ time if `C` is m-by-n in that case. It does not use any workspace at all. Since it uses no workspace, it can work very well for extremely sparse or hypersparse matrices, when the mask is present and not complemented.

- `GxB_NTHREADS` controls how many threads a method uses. By default (if set to zero, or `GxB_DEFAULT`), all available threads are used. The maximum available threads is controlled by the global setting, which is `omp_get_max_threads ( )` by default. If set to some positive integer `nthreads` less than this maximum, at most `nthreads` threads will be used. See Section 8.1 for details.
- `GxB_CHUNK` is a `double` value that controls how many threads a method uses for small problems. See Section 8.1 for details.
- `GxB_SORT` provides a hint to `GrB_mxm`, `GrB_mxv`, `GrB_vxm`, and `GrB_reduce` (to vector). These methods can leave the output matrix or vector in a jumbled state, where the final sort is left as pending work. This is typically fastest, since some algorithms can tolerate jumbled matrices on input, and sometimes the sort can be skipped entirely. However, if the matrix or vector will be immediately exported in unjumbled form, or provided as input to a method that requires it to not be jumbled, then sorting it during the matrix multiplication is faster. By default, these methods leave the result in jumbled form (a *lazy sort*), if `GxB_SORT` is set to zero (`GxB_DEFAULT`). A nonzero value will inform the matrix multiplication to sort its result, instead.
- `GxB_COMPRESSION` selects the compression method for serialization. The default is LZ4. See Section 6.10 for other options.
- `GxB_IMPORT` informs the `GxB` pack methods that they can trust their input data, or not. The default is to trust the input, for faster packing. If the data is being packed from an untrusted source, then additional checks should be made, and the following descriptor setting should be used:

```
GxB_set (desc, GxB_IMPORT, GxB_SECURE_IMPORT) ;
```

The next sections describe the methods for a `GrB_Descriptor`:

GraphBLAS function	purpose	Section
`GrB_Descriptor_new`	create a descriptor	6.14.1
`GrB_Descriptor_wait`	wait for a descriptor	6.14.2
`GrB_Descriptor_set`	set a parameter in a descriptor	6.14.3
`GxB_Desc_set`	set a parameter in a descriptor	6.14.4
`GxB_Desc_get`	get a parameter from a descriptor	6.14.5
`GrB_Descriptor_free`	free a descriptor	6.14.6

6.14.1 GrB_Descriptor_new: create a new descriptor

```
GrB_Info GrB_Descriptor_new     // create a new descriptor
(
    GrB_Descriptor *descriptor  // handle of descriptor to create
) ;
```

`GrB_Descriptor_new` creates a new descriptor, with all fields set to their defaults (output is not replaced, the mask is not complemented, the mask is valued not structural, neither input matrix is transposed, the method used in `C=A*B` is selected automatically, and `GrB_mxm` leaves the final sort as pending work).

6.14.2 GrB_Descriptor_wait: wait for a descriptor

```
// in SuiteSparse:GraphBLAS v5.x and earlier:
GrB_Info GrB_wait                   // wait for a descriptor
(
    GrB_Descriptor *descriptor      // descriptor to wait for
) ;

// in SuiteSparse:GraphBLAS v6 (v2.0 C API):
GrB_Info GrB_wait                   // wait for a descriptor
(
    GrB_Descriptor descriptor,      // descriptor to wait for
    GrB_WaitMode mode               // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

After creating a user-defined descriptor, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. Currently, SuiteSparse:GraphBLAS does nothing except to ensure that `d` is valid.

6.14.3 GrB_Descriptor_set: set a parameter in a descriptor

```
GrB_Info GrB_Descriptor_set     // set a parameter in a descriptor
(
    GrB_Descriptor desc,        // descriptor to modify
    GrB_Desc_Field field,       // parameter to change
    GrB_Desc_Value val          // value to change it to
) ;
```

`GrB_Descriptor_set` sets a descriptor field (`GrB_OUTP`, `GrB_MASK`, `GrB_INP0`, `GrB_INP1`, or `GxB_AxB_METHOD`) to a particular value. Use `GxB_Dec_set` to set the value of `GxB_NTHREADS`, `GxB_CHUNK`, and `GxB_SORT`. If an error occurs, `GrB_error(&err,desc)` returns details about the error.

Descriptor field	Default	Non-default
`GrB_OUTP`	`GxB_DEFAULT`: The output matrix is not cleared. The operation computes $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$.	`GrB_REPLACE`: After computing $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$, the output $\mathbf{C}$ is cleared of all entries. Then $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ is performed.
`GrB_MASK`	`GxB_DEFAULT`: The Mask is not complemented. `Mask(i,j)=1` means the value C_{ij} can be modified by the operation, while `Mask(i,j)=0` means the value C_{ij} shall not be modified by the operation.	`GrB_COMP`: The Mask is complemented. `Mask(i,j)=0` means the value C_{ij} can be modified by the operation, while `Mask(i,j)=1` means the value C_{ij} shall not be modified by the operation. `GrB_STRUCTURE`: The values of the Mask are ignored. If `Mask(i,j)` is an entry in the `Mask` matrix, it is treated as if `Mask(i,j)=1`. The two options `GrB_COMP` and `GrB_STRUCTURE` can be combined, with two subsequent calls, or with a single call with the setting `GrB_COMP+GrB_STRUCTURE`.
`GrB_INP0`	`GxB_DEFAULT`: The first input is not transposed prior to using it in the operation.	`GrB_TRAN`: The first input is transposed prior to using it in the operation. Only matrices are transposed, never vectors.
`GrB_INP1`	`GxB_DEFAULT`: The second input is not transposed prior to using it in the operation.	`GrB_TRAN`: The second input is transposed prior to using it in the operation. Only matrices are transposed, never vectors.
`GrB_AxB_METHOD`	`GxB_DEFAULT`: The method for `C=A*B` is selected automatically.	`GxB_AxB_`*method*: The selected method is used to compute `C=A*B`.

6.14.4 GxB_Desc_set: set a parameter in a descriptor

```
GrB_Info GxB_Desc_set           // set a parameter in a descriptor
(
    GrB_Descriptor desc,        // descriptor to modify
    GrB_Desc_Field field,       // parameter to change
    ...                         // value to change it to
) ;
```

`GxB_Desc_set` is like `GrB_Descriptor_set`, except that the type of the third parameter can vary with the field. This function can modify all descriptor settings, including those that do not have the type `GrB_Desc_Value`. See also `GxB_set` described in Section 8. If an error occurs, `GrB_error(&err,desc)` returns details about the error.

6.14.5 GxB_Desc_get: get a parameter from a descriptor

```
GrB_Info GxB_Desc_get           // get a parameter from a descriptor
(
    GrB_Descriptor desc,        // descriptor to query; NULL means defaults
    GrB_Desc_Field field,       // parameter to query
    ...                         // value of the parameter
) ;
```

`GxB_Desc_get` returns the value of a single field in a descriptor. The type of the third parameter is a pointer to a variable type, whose type depends on the field. See also `GxB_get` described in Section 8.

6.14.6 GrB_Descriptor_free: free a descriptor

```
GrB_Info GrB_free               // free a descriptor
(
    GrB_Descriptor *descriptor  // handle of descriptor to free
) ;
```

`GrB_Descriptor_free` frees a descriptor. Either usage:

```
GrB_Descriptor_free (&descriptor) ;
GrB_free (&descriptor) ;
```

frees the `descriptor` and sets `descriptor` to `NULL`. It safely does nothing if passed a `NULL` handle, or if `descriptor == NULL` on input.

6.14.7 GrB_DESC_*: built-in descriptors

Built-in descriptors are listed in the table below. A dash in the table indicates the default. These descriptors may not be modified or freed. Attempts to modify them result in an error (GrB_INVALID_VALUE); attempts to free them are silently ignored.

Descriptor	OUTP	MASK structural	MASK complement	INP0	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_CT1	-	-	GrB_COMP	-	GrB_TRAN
GrB_DESC_CT0	-	-	GrB_COMP	GrB_TRAN	-
GrB_DESC_CT0T1	-	-	GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_S	-	GrB_STRUCTURE	-	-	-
GrB_DESC_ST1	-	GrB_STRUCTURE	-	-	GrB_TRAN
GrB_DESC_ST0	-	GrB_STRUCTURE	-	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_RCT1	GrB_REPLACE	-	GrB_COMP	-	GrB_TRAN
GrB_DESC_RCT0	GrB_REPLACE	-	GrB_COMP	GrB_TRAN	-
GrB_DESC_RCT0T1	GrB_REPLACE	-	GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_RS	GrB_REPLACE	GrB_STRUCTURE	-	-	-
GrB_DESC_RST1	GrB_REPLACE	GrB_STRUCTURE	-	-	GrB_TRAN
GrB_DESC_RST0	GrB_REPLACE	GrB_STRUCTURE	-	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

6.15 GrB_free: free any GraphBLAS object

Each of the ten objects has `GrB_*_new` and `GrB_*_free` methods that are specific to each object. They can also be accessed by a generic function, `GrB_free`, that works for all ten objects. If `G` is any of the ten objects, the statement

```
GrB_free (&G) ;
```

frees the object and sets the variable `G` to `NULL`. It is safe to pass in a `NULL` handle, or to free an object twice:

```
GrB_free (NULL) ;       // SuiteSparse:GraphBLAS safely does nothing
GrB_free (&G) ;         // the object G is freed and G set to NULL
GrB_free (&G) ;         // SuiteSparse:GraphBLAS safely does nothing
```

However, the following sequence of operations is not safe. The first two are valid but the last statement will lead to undefined behavior.

```
H = G ;                 // valid; creates a 2nd handle of the same object
GrB_free (&G) ;         // valid; G is freed and set to NULL; H now undefined
GrB_some_method (H) ;   // not valid; H is undefined
```

Some objects are predefined, such as the built-in types. If a user application attempts to free a built-in object, SuiteSparse:GraphBLAS will safely do nothing. The `GrB_free` function in SuiteSparse:GraphBLAS always returns `GrB_SUCCESS`.

7 The mask, accumulator, and replace option

After a GraphBLAS operation computes a result $\mathbf{T}$, (for example, $\mathbf{T} = \mathbf{AB}$ for `GrB_mxm`), the results are assigned to an output matrix $\mathbf{C}$ via the mask/ accumulator phase, written as $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$. This phase is affected by the `GrB_REPLACE` option in the descriptor, the presence of an optional binary accumulator operator ($\odot$), the presence of the optional mask matrix $\mathbf{M}$, and the status of the mask descriptor. The interplay of these options is summarized in Table 1.

The mask $\mathbf{M}$ may be present, or not. It may be structural or valued, and it may be complemented, or not. These options may be combined, for a total of 8 cases, although the structural/valued option as no effect if $\mathbf{M}$ is not present. If $\mathbf{M}$ is not present and not complemented, then m_{ij} is implicitly true. If not present yet complemented, then all m_{ij} entries are implicitly zero; in this case, $\mathbf{T}$ need not be computed at all. Either $\mathbf{C}$ is not modified, or all its entries are cleared if the replace option is enabled. If $\mathbf{M}$ is present, and the structural option is used, then m_{ij} is treated as true if it is an entry in the matrix (its value is ignored). Otherwise, the value of m_{ij} is used. In both cases, entries not present are implicitly zero. These values are negated if the mask is complemented. All of these various cases are combined to give a single effective value of the mask at position ij.

The combination of all these options are presented in the Table 1. The first column is the `GrB_REPLACE` option. The second column lists whether or not the accumulator operator is present. The third column lists whether or not c_{ij} exists on input to the mask/accumulator phase (a dash means that it does not exist). The fourth column lists whether or not the entry t_{ij} is present in the result matrix $\mathbf{T}$. The mask column is the final effective value of m_{ij}, after accounting for the presence of $\mathbf{M}$ and the mask options. Finally, the last column states the result of the mask/accum step; if no action is listed in this column, then c_{ij} is not modified.

Several important observations can be made from this table. First, if no mask is present (and the mask-complement descriptor option is not used), then only the first half of the table is used. In this case, the `GrB_REPLACE` option has no effect. The entire matrix $\mathbf{C}$ is modified.

Consider the cases when c_{ij} is present but t_{ij} is not, and there is no mask or the effective value of the mask is true for this ij position. With no accumulator operator, c_{ij} is deleted. If the accumulator operator is present and the replace option is not used, c_{ij} remains unchanged.

When there is no mask and the mask `GrB_COMP` option is not selected, the table simplifies (Table 2). The `GrB_REPLACE` option no longer has any effect. The `GrB_SECOND_T` binary operator when used as the accumulator unifies the first cases, shown in Table 3. The only difference now is the behavior when c_{ij} is present but t_{ij} is not. Finally, the effect of `GrB_FIRST_T` as the accumulator is shown in Table 4.

repl	accum	$\mathbf{C}$	$\mathbf{T}$	mask	action taken by $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$
-	-	c_{ij}	t_{ij}	1	$c_{ij} = t_{ij}$, update
-	-	-	t_{ij}	1	$c_{ij} = t_{ij}$, insert
-	-	c_{ij}	-	1	delete c_{ij} because t_{ij} not present
-	-	-	-	1	
-	-	c_{ij}	t_{ij}	0	
-	-	-	t_{ij}	0	
-	-	c_{ij}	-	0	
-	-	-	-	0	
yes	-	c_{ij}	t_{ij}	1	$c_{ij} = t_{ij}$, update
yes	-	-	t_{ij}	1	$c_{ij} = t_{ij}$, insert
yes	-	c_{ij}	-	1	delete c_{ij} because t_{ij} not present
yes	-	-	-	1	
yes	-	c_{ij}	t_{ij}	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	t_{ij}	0	
yes	-	c_{ij}	-	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	-	0	
-	yes	c_{ij}	t_{ij}	1	$c_{ij} = c_{ij} \odot t_{ij}$, apply accumulator
-	yes	-	t_{ij}	1	$c_{ij} = t_{ij}$, insert
-	yes	c_{ij}	-	1	
-	yes	-	-	1	
-	yes	c_{ij}	t_{ij}	0	
-	yes	-	t_{ij}	0	
-	yes	c_{ij}	-	0	
-	yes	-	-	0	
yes	yes	c_{ij}	t_{ij}	1	$c_{ij} = c_{ij} \odot t_{ij}$, apply accumulator
yes	yes	-	t_{ij}	1	$c_{ij} = t_{ij}$, insert
yes	yes	c_{ij}	-	1	
yes	yes	-	-	1	
yes	yes	c_{ij}	t_{ij}	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	t_{ij}	0	
yes	yes	c_{ij}	-	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	-	0	

Table 1: Results of the mask/accumulator phase.

accum	$\mathbf{C}$	$\mathbf{T}$	action taken by $\mathbf{C} = \mathbf{C} \odot \mathbf{T}$
-	c_{ij}	t_{ij}	$c_{ij} = t_{ij}$, update
-	-	t_{ij}	$c_{ij} = t_{ij}$, insert
-	c_{ij}	-	delete c_{ij} because t_{ij} not present
-	-	-	
yes	c_{ij}	t_{ij}	$c_{ij} = c_{ij} \odot t_{ij}$, apply accumulator
yes	-	t_{ij}	$c_{ij} = t_{ij}$, insert
yes	c_{ij}	-	
yes	-	-	

Table 2: When no mask is present (and not complemented).

accum	$\mathbf{C}$	$\mathbf{T}$	action taken by $\mathbf{C} = \mathbf{C} \odot \mathbf{T}$
yes	c_{ij}	t_{ij}	$c_{ij} = t_{ij}$, apply `GrB_SECOND` accumulator
yes	-	t_{ij}	$c_{ij} = t_{ij}$, insert
yes	c_{ij}	-	
yes	-	-	

Table 3: No mask, with the SECOND operator as the accumulator.

8 SuiteSparse:GraphBLAS Options

SuiteSparse:GraphBLAS includes two type-generic methods, `GxB_set` and `GxB_get`, that set and query various options and parameters settings, including a generic way to set values in the `GrB_Descriptor` object. Using these methods, the user application can provide hints to SuiteSparse:GraphBLAS on how it should store and operate on its matrices. These hints have no effect on the results of any GraphBLAS operation (except perhaps floating-point roundoff differences), but they can have a great impact on the amount of time or memory taken.

- `GxB_set (field, value)` sets global options.

accum	$\mathbf{C}$	$\mathbf{T}$	action taken by $\mathbf{C} = \mathbf{C} \odot \mathbf{T}$
yes	c_{ij}	t_{ij}	
yes	-	t_{ij}	$c_{ij} = t_{ij}$, insert
yes	c_{ij}	-	
yes	-	-	

Table 4: No Mask, with the FIRST operator as the accumulator.

field	value	description
`GxB_HYPER_SWITCH`	`double`	hypersparsity control (0 to 1)
`GxB_BITMAP_SWITCH`	`double [8]`	bitmap control
`GxB_FORMAT`	`int`	`GxB_BY_ROW` or `GxB_BY_COL`
`GxB_GLOBAL_NTHREADS`	`int`	number of threads to use
`GxB_NTHREADS`	`int`	number of threads to use
`GxB_GLOBAL_CHUNK`	`double`	chunk size
`GxB_CHUNK`	`double`	chunk size
`GxB_BURBLE`	`int`	diagnostic output
`GxB_PRINTF`	see below	diagnostic output
`GxB_FLUSH`	see below	diagnostic output
`GxB_MEMORY_POOL`	`int64_t [64]`	memory pool control
`GxB_PRINT_1BASED`	`int`	for printing matrices/vectors

- `GxB_set (GrB_Matrix A, field, value)` provides hints to SuiteSparse: GraphBLAS on how to store a particular matrix.

field	value	description
`GxB_HYPER_SWITCH`	`double`	hypersparsity control (0 to 1)
`GxB_BITMAP_SWITCH`	`double`	bitmap control (0 to 1)
`GxB_FORMAT`	`int`	`GxB_BY_ROW` or `GxB_BY_COL`
`GxB_SPARSITY_CONTROL`	`int`	0 to 15

- `GxB_set (GrB_Vector v, field, value)` provides hints to SuiteSparse: GraphBLAS on how to store a particular vector.

field	value	description
`GxB_BITMAP_SWITCH`	`double`	bitmap control (0 to 1)
`GxB_SPARSITY_CONTROL`	`int`	0 to 15

- `GxB_set (GrB_Descriptor desc, field, value)` sets the value of a field in a `GrB_Descriptor`.

field	value	description
GrB_OUTP	GrB_Desc_Value	replace option
GrB_MASK	GrB_Desc_Value	mask option
GrB_INP0	GrB_Desc_Value	transpose input 0
GrB_INP1	GrB_Desc_Value	transpose input 1
GxB_DESCRIPTOR_NTHREADS	int	number of threads to use
GxB_NTHREADS	int	number of threads to use
GxB_DESCRIPTOR_CHUNK	double	chunk size
GxB_CHUNK	double	chunk size
GxB_AxB_METHOD	int	method for matrix multiply
GxB_SORT	int	lazy vs aggressive sort
GxB_COMPRESSION	int	compression for serialization
GxB_IMPORT	GrB_Desc_Value	trust data on import/pack

GxB_get queries a GrB_Descriptor, a GrB_Matrix, a GrB_Vector, or the global options.

- GxB_get (field, &value) retrieves the value of a global option.

field	value	description
GxB_HYPER_SWITCH	double	hypersparsity control (0 to 1)
GxB_BITMAP_SWITCH	double [8]	bitmap control
GxB_FORMAT	int	GxB_BY_ROW or GxB_BY_COL
GxB_GLOBAL_NTHREADS	int	number of threads to use
GxB_NTHREADS	int	number of threads to use
GxB_GLOBAL_CHUNK	double	chunk size
GxB_CHUNK	double	chunk size
GxB_BURBLE	int	diagnostic output
GxB_PRINTF	see below	diagnostic output
GxB_FLUSH	see below	diagnostic output
GxB_MEMORY_POOL	int64_t [64]	memory pool control
GxB_PRINT_1BASED	int	for printing matrices/vectors
GxB_MODE	int	blocking/non-blocking
GxB_LIBRARY_NAME	char *	name of library
GxB_LIBRARY_VERSION	int [3]	library version
GxB_LIBRARY_DATE	char *	release date
GxB_LIBRARY_ABOUT	char *	about the library
GxB_LIBRARY_LICENSE	char *	license
GxB_LIBRARY_COMPILE_DATE	char *	date of compilation
GxB_LIBRARY_COMPILE_TIME	char *	time of compilation
GxB_LIBRARY_URL	char *	url of library
GxB_API_VERSION	int [3]	C API version
GxB_API_DATE	char *	C API date
GxB_API_ABOUT	char *	about the C API
GxB_API_URL	char *	http://graphblas.org
GxB_COMPILER_NAME	char *	C compiler name
GxB_COMPILER_VERSION	int [3]	C compiler version

- `GxB_get (GrB_Matrix A, field, &value)` retrieves the current value of an option from a particular matrix `A`.

field	value	description
`GxB_HYPER_SWITCH`	`double`	hypersparsity control (0 to 1)
`GxB_BITMAP_SWITCH`	`double`	bitmap control (0 to 1)
`GxB_FORMAT`	`int`	`GxB_BY_ROW` or `GxB_BY_COL`
`GxB_SPARSITY_CONTROL`	`int`	0 to 15
`GxB_SPARSITY_STATUS`	`int`	1, 2, 4, or 8

- `GxB_get (GrB_Vector A, field, &value)` retrieves the current value of an option from a particular vector `v`.

field	value	description
`GxB_BITMAP_SWITCH`	`double`	bitmap control (0 to 1)
`GxB_FORMAT`	`int`	`GxB_BY_ROW` or `GxB_BY_COL`
`GxB_SPARSITY_CONTROL`	`int`	0 to 15
`GxB_SPARSITY_STATUS`	`int`	1, 2, 4, or 8

- `GxB_get (GrB_Descriptor desc, field, &value)` retrieves the value of a field in a descriptor.

field	value	description
`GrB_OUTP`	`GrB_Desc_Value`	replace option
`GrB_MASK`	`GrB_Desc_Value`	mask option
`GrB_INP0`	`GrB_Desc_Value`	transpose input 0
`GrB_INP1`	`GrB_Desc_Value`	transpose input 1
`GxB_DESCRIPTOR_NTHREADS`	`int`	number of threads to use
`GxB_NTHREADS`	`int`	number of threads to use
`GxB_DESCRIPTOR_CHUNK`	`double`	chunk size
`GxB_CHUNK`	`double`	chunk size
`GxB_AxB_METHOD`	`int`	method for matrix multiply
`GxB_SORT`	`int`	lazy vs aggressive sort
`GxB_COMPRESSION`	`int`	compression for serialization
`GxB_IMPORT`	`GrB_Desc_Value`	trust data on import/pack

8.1 OpenMP parallelism

SuiteSparse:GraphBLAS is a parallel library, based on OpenMP. By default, all GraphBLAS operations will use up to the maximum number of threads specified by the `omp_get_max_threads` OpenMP function. For small problems, GraphBLAS may choose to use fewer threads, using two parameters: the maximum number of threads to use (which may differ from the `omp_get_max_threads` value), and a parameter called the `chunk`. Suppose `work` is a measure of the work an operation needs to perform (say the number of entries in the two input matrices for `GrB_eWiseAdd`). No more than `floor(work/chunk)` threads will be used (or one thread if the ratio is less than 1).

The default `chunk` value is 65,536, but this may change in future versions, or it may be modified when GraphBLAS is installed on a particular machine.

Both parameters can be set in two ways:

- Globally: If the following methods are used, then all subsequent GraphBLAS operations will use these settings. Note the typecast, `(double) chunk`. This is necessary if a literal constant such as `20000` is passed as this argument. The type of the constant must be `double`.

```
int nthreads_max = 40 ;
GxB_set (GxB_NTHREADS, nthreads_max) ;
GxB_set (GxB_CHUNK, (double) 20000) ;
```

- Per operation: Most GraphBLAS operations take a `GrB_Descriptor` input, and this can be modified to set the number of threads and chunk size for the operation that uses this descriptor. Note that `chunk` is a `double`.

```
GrB_Descriptor desc ;
GrB_Descriptor_new (&desc)
int nthreads_max = 40 ;
GxB_set (desc, GxB_NTHREADS, nthreads_max) ;
double chunk = 20000 ;
GxB_set (desc, GxB_CHUNK, chunk) ;
```

The smaller of `nthreads_max` and `floor(work/chunk)` is used for any given GraphBLAS operation, except that a single thread is used if this value is zero or less.

If either parameter is set to `GxB_DEFAULT`, then default values are used. The default for `nthreads_max` is the return value from `omp_get_max_threads`, and the default chunk size is currently 65,536.

If a descriptor value for either parameter is left at its default, or set to `GxB_DEFAULT`, then the global setting is used. This global setting may have been modified from its default, and this modified value will be used.

For example, suppose `omp_get_max_threads` reports 8 threads. If `GxB_set (GxB_NTHREADS, 4)` is used, then the global setting is four threads, not eight. If a descriptor is used but its `GxB_NTHREADS` is not set, or set to `GxB_DEFAULT`, then any operation that uses this descriptor will use 4 threads.

8.2 Storing a matrix by row or by column

The GraphBLAS `GrB_Matrix` is entirely opaque to the user application, and the GraphBLAS API does not specify how the matrix should be stored. However, choices made in how the matrix is represented in a particular implementation, such as SuiteSparse:GraphBLAS, can have a large impact on performance.

Many graph algorithms are just as fast in any format, but some algorithms are much faster in one format or the other. For example, suppose the user application stores a directed graph as a matrix `A`, with the edge (i, j) represented as the value `A(i,j)`, and the application makes many accesses to the ith row of the matrix, with `GrB_Col_extract (w,...,A,GrB_ALL,...,i,desc)` with the transposed descriptor (`GrB_INP0` set to `GrB_TRAN`). If the matrix is stored by column this can be extremely slow, just like the expression

`w=A(i,:)` in MATLAB, where `i` is a scalar. Since this is a typical use-case in graph algorithms, the default format in SuiteSparse:GraphBLAS is to store its matrices by row, in Compressed Sparse Row format (CSR).

MATLAB stores its sparse matrices by column, in "non-hypersparse" format, in what is called the Compressed Sparse Column format, or CSC for short. An `m`-by-`n` matrix in MATLAB is represented as a set of `n` column vectors, each with a sorted list of row indices and values of the nonzero entries in that column. As a result, `w=A(:,j)` is very fast in MATLAB, since the result is already held in the data structure a single list, the jth column vector. However, `w=A(i,:)` is very slow in MATLAB, since every column in the matrix has to be searched to see if it contains row `i`. In MATLAB, if many such accesses are made, it is much better to transpose the matrix (say `AT=A'`) and then use `w=AT(:,i)` instead. This can have a dramatic impact on the performance of MATLAB.

Likewise, if `u` is a very sparse column vector and `A` is stored by column, then `w=u'*A` (via `GrB_vxm`) is slower than `w=A*u` (via `GrB_mxv`). The opposite is true if the matrix is stored by row.

SuiteSparse:GraphBLAS stores its matrices by row, by default (with one exception described below). However, it can also be instructed to store any selected matrices, or all matrices, by column instead (just like MATLAB), so that `w=A(:,j)` (via `GrB_Col_extract`) is very fast. The change in data format has no effect on the result, just the time and memory usage. To use a column-oriented format by default, the following can be done in a user application that tends to access its matrices by column.

```
GrB_init (...) ;
// just after GrB_init: do the following:
#ifdef GxB_SUITESPARSE_GRAPHBLAS
GxB_set (GxB_FORMAT, GxB_BY_COL) ;
#endif
```

If this is done, and no other `GxB_set` calls are made with `GxB_FORMAT`, all matrices will be stored by column. The default format is `GxB_BY_ROW`.

All vectors (`GrB_Vector`) are held by column, and this cannot be changed.

By default, matrices of size `m-by-1` are held by column, regardless of the global setting described above. Matrices of size `1-by-n` with `n` not equal to 1 are held by row, regardless of the global setting. The global setting only affects matrices with both `m > 1` and `n > 1`. Empty matrices (`0-by-0`) are also controlled by the global setting.

After creating a matrix with `GrB_Matrix_new (&A, ...)`, its format can be changed arbitrarily with `GxB_set (A, GxB_FORMAT, ...)`. So even an `m-by-1` matrix can then be changed to be held by row, for example. Likewise, once a `1-by-n` matrix is created, it can be converted to column-oriented format.

8.3 Hypersparse matrices

MATLAB can store an `m`-by-`n` matrix with a very large value of `m`, since a CSC data structure takes $O(n+|\mathbf{A}|)$ memory, independent of `m`, where $|\mathbf{A}|$ is the number of nonzeros in the matrix. It cannot store a matrix with a huge `n`, and this structure is also inefficient

when $|\mathbf{A}|$ is much smaller than `n`. In contrast, SuiteSparse:GraphBLAS can store its matrices in *hypersparse* format, taking only $O(|\mathbf{A}|)$ memory, independent of how it is stored (by row or by column) and independent of both `m` and `n` [BG08, BG12].

In both the CSR and CSC formats, the matrix is held as a set of sparse vectors. In non-hypersparse format, the set of sparse vectors is itself dense; all vectors are present, even if they are empty. For example, an `m`-by-`n` matrix in non-hypersparse CSC format contains `n` sparse vectors. Each column vector takes at least one integer to represent, even for a column with no entries. This allows for quick lookup for a particular vector, but the memory required is $O(n + |\mathbf{A}|)$. With a hypersparse CSC format, the set of vectors itself is sparse, and columns with no entries take no memory at all. The drawback of the hypersparse format is that finding an arbitrary column vector `j`, such as for the computation `C=A(:,j)`, takes $O(\log k)$ time if there $k \leq n$ vectors in the data structure. One advantage of the hypersparse structure is the memory required for an `m`-by-`n` hypersparse CSC matrix is only $O(|\mathbf{A}|)$, independent of `m` and `n`. Algorithms that must visit all non-empty columns of a matrix are much faster when working with hypersparse matrices, since empty columns can be skipped.

The `hyper_switch` parameter controls the hypersparsity of the internal data structure for a matrix. The parameter is typically in the range 0 to 1. The default is `hyper_switch` = `GxB_HYPER_DEFAULT`, which is an `extern const double` value, currently set to 0.0625, or 1/16. This default ratio may change in the future.

The `hyper_switch` determines how the matrix is converted between the hypersparse and non-hypersparse formats. Let n be the number of columns of a CSC matrix, or the number of rows of a CSR matrix. The matrix can have at most n non-empty vectors.

Let k be the actual number of non-empty vectors. That is, for the CSC format, $k \leq n$ is the number of columns that have at least one entry. Let h be the value of `hyper_switch`.

If a matrix is currently hypersparse, it can be converted to non-hypersparse if the either condition $n \leq 1$ or $k > 2nh$ holds, or both. Otherwise, it stays hypersparse. Note that if $n \leq 1$ the matrix is always stored as non-hypersparse.

If currently non-hypersparse, it can be converted to hypersparse if both conditions $n > 1$ and $k \leq nh$ hold. Otherwise, it stays non-hypersparse. Note that if $n \leq 1$ the matrix always remains non-hypersparse.

The default value of `hyper_switch` is assigned at startup by `GrB_init`, and can then be modified globally with `GxB_set`. All new matrices are created with the same `hyper_switch`, determined by the global value. Once a particular matrix `A` has been constructed, its hypersparsity ratio can be modified from the default with:

```
double hyper_switch = 0.2 ;
GxB_set (A, GxB_HYPER_SWITCH, hyper_switch) ;
```

To force a matrix to always be non-hypersparse, use `hyper_switch` equal to `GxB_NEVER_HYPER`. To force a matrix to always stay hypersparse, set `hyper_switch` to `GxB_ALWAYS_HYPER`.

A `GrB_Matrix` can thus be held in one of four formats: any combination of hyper/non-hyper and CSR/CSC. All `GrB_Vector` objects are always stored in non-hypersparse CSC format.

A new matrix created via `GrB_Matrix_new` starts with $k = 0$ and is created in hypersparse form by default unless $n \leq 1$ or if $h < 0$, where h is the global `hyper_switch` value.

The matrix is created in either `GxB_BY_ROW` or `GxB_BY_COL` format, as determined by the last call to `GxB_set(GxB_FORMAT,...)` or `GrB_init`.

A new matrix `C` created via `GrB_dup (&C,A)` inherits the CSR/CSC format, hypersparsity format, and `hyper_switch` from `A`.

8.4 Bitmap matrices

By default, SuiteSparse:GraphBLAS switches between all four formats (hypersparse, sparse, bitmap, and full) automatically. Let $d = |\mathbf{A}|/mn$ for an m-by-n matrix $\mathbf{A}$ with $|\mathbf{A}|$ entries. If the matrix is currently in sparse or hypersparse format, and is modified so that d exceeds a given threshold, it is converted into bitmap format. The default threshold is controlled by the `GxB_BITMAP_SWITCH` setting, which can be set globally, or for a particular matrix or vector.

The default value of the switch to bitmap format depends on $\min(m, n)$, for a matrix of size m-by-n. For the global setting, the bitmap switch is a `double` array of size `GxB_NBITMAP_SWITCH`. The defaults are given below:

parameter	default	matrix sizes
`bitmap_switch [0]`	0.04	$\min(m, n) = 1$ (and all vectors)
`bitmap_switch [1]`	0.05	$\min(m, n) = 2$
`bitmap_switch [2]`	0.06	$\min(m, n) = 3$ to 4
`bitmap_switch [3]`	0.08	$\min(m, n) = 5$ to 8
`bitmap_switch [4]`	0.10	$\min(m, n) = 9$ to 16
`bitmap_switch [5]`	0.20	$\min(m, n) = 17$ to 32
`bitmap_switch [6]`	0.30	$\min(m, n) = 33$ to 64
`bitmap_switch [7]`	0.40	$\min(m, n) > 64$

That is, by default a `GrB_Vector` is held in bitmap format if its density exceeds 4%. To change the global settings, do the following:

```
double bswitch [GxB_NBITMAP_SWITCH] = { 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 } ;
GxB_set (GxB_BITMAP_SWITCH, bswitch) ;
```

If the matrix is currently in bitmap format, it is converted to full if all entries are present, or to sparse/hypersparse if d drops below $b/2$, if its bitmap switch is b. A matrix or vector with d between $b/2$ and b remains in its current format.

8.5 Parameter types

The `GxB_Option_Field` enumerated type gives the type of the `field` parameter for the second argument of `GxB_set` and `GxB_get`, for setting global options or matrix options.

```
typedef enum
```

```
{
    // for matrix/vector get/set and global get/set:
    GxB_HYPER_SWITCH = 0,    // defines switch to hypersparse (double value)
    GxB_BITMAP_SWITCH = 34,  // defines switch to hypersparse (double value)
    GxB_FORMAT = 1,     // defines CSR/CSC format: GxB_BY_ROW or GxB_BY_COL
    GxB_SPARSITY_CONTROL = 32,  // control the sparsity of a matrix or vector

    // for global get/set only:
    GxB_GLOBAL_NTHREADS = GxB_NTHREADS, // max number of threads to use
    GxB_GLOBAL_CHUNK = GxB_CHUNK,       // chunk size for small problems
    GxB_BURBLE = 99,                    // diagnositic output
    GxB_PRINTF = 101,               // printf function for diagnostic output
    GxB_FLUSH = 102,                // flush function for diagnostic output
    GxB_MEMORY_POOL = 103,  // memory pool control
    GxB_PRINT_1BASED = 104, // print matrices as 0-based or 1-based

    // for matrix/vector get only:
    GxB_SPARSITY_STATUS = 33,   // query the sparsity of a matrix or vector

    // for global get only:
    GxB_MODE = 2,       // mode passed to GrB_init (blocking or non-blocking)
    GxB_LIBRARY_NAME = 8,           // name of the library (char *)
    GxB_LIBRARY_VERSION = 9,        // library version (3 int's)
    GxB_LIBRARY_DATE = 10,          // date of the library (char *)
    GxB_LIBRARY_ABOUT = 11,         // about the library (char *)
    GxB_LIBRARY_URL = 12,           // URL for the library (char *)
    GxB_LIBRARY_LICENSE = 13,       // license of the library (char *)
    GxB_LIBRARY_COMPILE_DATE = 14,  // date library was compiled (char *)
    GxB_LIBRARY_COMPILE_TIME = 15,  // time library was compiled (char *)
    GxB_API_VERSION = 16,           // API version (3 int's)
    GxB_API_DATE = 17,              // date of the API (char *)
    GxB_API_ABOUT = 18,             // about the API (char *)
    GxB_API_URL = 19,               // URL for the API (char *)
}
GxB_Option_Field ;
```

The `GxB_FORMAT` field can be by row or by column, set to a value with the type `GxB_Format_Value`:

```
typedef enum
{
    GxB_BY_ROW = 0,     // CSR: compressed sparse row format
    GxB_BY_COL = 1      // CSC: compressed sparse column format
}
GxB_Format_Value ;
```

The default format is given by the predefined value GxB_FORMAT_DEFAULT, which is equal to GxB_BY_ROW. The default hypersparsity ratio is 0.0625 (1/16), but this value may change in the future.

Setting the GxB_HYPER_SWITCH field to GxB_ALWAYS_HYPER ensures a matrix always stays hypersparse. If set to GxB_NEVER_HYPER, it always stays non-hypersparse. At startup, GrB_init defines the following initial settings:

```
GxB_set (GxB_HYPER_SWITCH, GxB_HYPER_DEFAULT) ;
GxB_set (GxB_FORMAT, GxB_BY_ROW) ;
```

That is, by default, all new matrices are held by row in CSR format (except for n-by-1 matrices; see GrB_Matrix_new). If a matrix has fewer than $n/16$ columns, it can be converted to hypersparse format. If it has more than $n/8$ columns, it can be converted to non-hypersparse format. These options can be changed for all future matrices with GxB_set. For example, to change all future matrices to be in non-hypersparse CSC when created, use:

```
GxB_set (GxB_HYPER_SWITCH, GxB_NEVER_HYPER) ;
GxB_set (GxB_FORMAT, GxB_BY_COL) ;
```

Then if a particular matrix needs a different format, then (as an example):

```
GxB_set (A, GxB_HYPER_SWITCH, 0.1) ;
GxB_set (A, GxB_FORMAT, GxB_BY_ROW) ;
```

This changes the matrix A so that it is stored by row, and it is converted from non-hypersparse to hypersparse format if it has fewer than 10% non-empty columns. If it is hypersparse, it is a candidate for conversion to non-hypersparse if has 20% or more non-empty columns. If it has between 10% and 20% non-empty columns, it remains in its current format. MATLAB only supports a non-hypersparse CSC format. The format in SuiteSparse:GraphBLAS that is equivalent to the MATLAB format is:

```
GrB_init (...) ;
GxB_set (GxB_HYPER_SWITCH, GxB_NEVER_HYPER) ;
GxB_set (GxB_FORMAT, GxB_BY_COL) ;
// no subsequent use of GxB_HYPER_SWITCH or GxB_FORMAT
```

The GxB_HYPER_SWITCH and GxB_FORMAT options should be considered as suggestions from the user application as to how SuiteSparse:GraphBLAS can obtain the best performance for a particular application. SuiteSparse:GraphBLAS is free to ignore any of these suggestions, both now and in the future, and the available options and formats may be augmented in the future. Any prior options no longer needed in future versions of SuiteSparse:GraphBLAS will be silently ignored, so the use these options is safe for future updates.

The sparsity status of a matrix can be queried with the following, which returns a value of GxB_HYPERSPARSE GxB_SPARSE GxB_BITMAP or GxB_FULL.

```
int sparsity ;
GxB_get (A, GxB_SPARSITY_STATUS, &sparsity) ;
```

The sparsity format of a matrix can be controlled with `GxB_set`, which can be any mix (a sum or bitwise or) of `GxB_HYPERSPARSE GxB_SPARSE GxB_BITMAP`, and `GxB_FULL`. By default, a matrix or vector can be held in any format, with the default setting `GxB_AUTO_SPARSITY`, which is equal to `GxB_HYPERSPARSE + GxB_SPARSE + GxB_BITMAP + GxB_FULL`. To enable a matrix to take on just `GxB_SPARSE` or `GxB_FULL` formats, but not `GxB_HYPERSPARSE` or `GxB_BITMAP`, for example, use the following:

```
GxB_set (A, GxB_SPARSITY_CONTROL, GxB_SPARSE + GxB_FULL) ;
```

In this case, SuiteSparse:GraphBLAS will hold the matrix in sparse format (`CSC` or `CSC`, depending on its `GxB_FORMAT`), unless all entries are present, in which case it will be converted to full format.

Only the least 4 bits of the sparsity control are considered, so the formats can be bitwise negated. For example, to allow for any format except full:

```
GxB_set (A, GxB_SPARSITY_CONTROL, ~GxB_FULL) ;
```

8.6 GxB_BURBLE, GxB_PRINTF, GxB_FLUSH: diagnostics

`GxB_set (GxB_BURBLE, ...)` controls the burble setting. It can also be controlled via `GrB.burble(b)` in the Octave/MATLAB interface.

```
GxB_set (GxB_BURBLE, true) ;   // enable burble
GxB_set (GxB_BURBLE, false) ;  // disable burble
```

If enabled, SuiteSparse:GraphBLAS reports which internal kernels it uses, and how much time is spent. If you see the word `generic`, it means that SuiteSparse:GraphBLAS was unable to use is faster kernels in `Source/Generated2`, but used a generic kernel that relies on function pointers. This is done for user-defined types and operators, and when typecasting is performed, and it is typically slower than the kernels in `Source/Generated2`.

If you see a lot of `wait` statements, it may mean that a lot of time is spent finishing a matrix or vector. This may be the result of an inefficient use of the `setElement` and `assign` methods. If this occurs you might try changing the sparsity format of a vector or matrix to `GxB_BITMAP`, assuming there's enough space for it.

`GxB_set (GxB_PRINTF, printf)` allows the user application to change the function used to print diagnostic output. This also controls the output of the `GxB_*print` functions. By default this parameter is `NULL`, in which case the ANSI C11 `printf` function is used. The parameter is a function pointer with the same signature as the ANSI C11 `printf` function. The Octave/MATLAB interface to GraphBLAS uses the following so that GraphBLAS can print to the Octave/MATLAB Command Window:

```
GxB_set (GxB_PRINTF, mexPrintf)
```

After each call to the `printf` function, an optional `flush` function is called, which is `NULL` by default. If `NULL`, the function is not used. This can be changed with `GxB_set (GxB_FLUSH, flush)`. The `flush` function takes no arguments, and returns an `int` which is 0 if successful, or any nonzero value on failure (the same output as the ANSI C11 `fflush` function, except that `flush` has no inputs).

8.7 Other global options

`GxB_MODE` can only be queried by `GxB_get`; it cannot be modified by `GxB_set`. The mode is the value passed to `GrB_init` (blocking or non-blocking).

All threads in the same user application share the same global options, including hypersparsity, bitmap options, and CSR/CSC format determined by `GxB_set`, and the blocking mode determined by `GrB_init`. Specific format and hypersparsity parameters of each matrix are specific to that matrix and can be independently changed.

The `GxB_LIBRARY_*` options can be used with `GxB_get` to query the current implementation. For all of these, `GxB_get` returns a string (`char *`), except for `GxB_LIBRARY_VERSION`, which takes as input an `int` array of size three. The `GxB_API_*` options can be used with `GxB_get` to query the current GraphBLAS C API Specification. For all of these, `GxB_get` returns a string (`char *`), except for `GxB_API_VERSION`, which takes as input an `int` array of size three.

8.8 GxB_Global_Option_set: set a global option

```
GrB_Info GxB_set                    // set a global default option
(
    const GxB_Option_Field field,   // option to change
    ...                             // value to change it to
) ;
```

This usage of `GxB_set` sets the value of a global option. The `field` parameter can be `GxB_HYPER_SWITCH`, `GxB_BITMAP_SWITCH`, `GxB_FORMAT`, `GxB_NTHREADS`, `GxB_CHUNK`, `GxB_BURBLE`, `GxB_PRINTF`, `GxB_FLUSH`, `GxB_MEMORY_POOL`, or `GxB_PRINT_1BASED`.

For example, the following usage sets the global hypersparsity ratio to 0.2, the format of future matrices to `GxB_BY_COL`, the maximum number of threads to 4, the chunk size to 10000, and enables the burble. No existing matrices are changed.

```
GxB_set (GxB_HYPER_SWITCH, 0.2) ;
GxB_set (GxB_FORMAT, GxB_BY_COL) ;
GxB_set (GxB_NTHREADS, 4) ;
GxB_set (GxB_CHUNK, (double) 10000) ;
GxB_set (GxB_BURBLE, true) ;
GxB_set (GxB_PRINTF, mexPrintf) ;
```

The memory pool parameter sets an upper bound on the number of freed blocks of memory that SuiteSparse:GraphBLAS keeps in its internal memory pool for future

allocations. `free_pool_limit` is an `int64_t` array of size 64, and `free_pool_limit [k]` is the upper bound on the number of blocks of size 2^k that are kept in the pool. Passing in a `NULL` pointer sets the defaults. Passing in an array of size 64 whose entries are all zero disables the memory pool entirely.

8.9 GxB_Matrix_Option_set: set a matrix option

```
GrB_Info GxB_set                    // set an option in a matrix
(
    GrB_Matrix A,                   // matrix to modify
    const GxB_Option_Field field,   // option to change
    ...                             // value to change it to
) ;
```

This usage of `GxB_set` sets the value of a matrix option, for a particular matrix. The `field` parameter can be `GxB_HYPER_SWITCH`, `GxB_BITMAP_SWITCH`, `GxB_SPARSITY_CONTROL`, or `GxB_FORMAT`.

For example, the following usage sets the hypersparsity ratio to 0.2, and the format of `GxB_BY_COL`, for a particular matrix `A`, and sets the sparsity control to `GxB_SPARSE+GxB_FULL` (allowing the matrix to be held in CSC or FullC formats, but not BitmapC or HyperCSC). SuiteSparse:GraphBLAS currently applies these changes immediately, but since they are simply hints, future versions of SuiteSparse:GraphBLAS may delay the change in format if it can obtain better performance.

If the setting is just `GxB_FULL` and some entries are missing, then the matrix is held in bitmap format.

```
GxB_set (A, GxB_HYPER_SWITCH, 0.2) ;
GxB_set (A, GxB_FORMAT, GxB_BY_COL) ;
GxB_set (A, GxB_SPARSITY_CONTROL, GxB_SPARSE + GxB_FULL) ;
```

For performance, the matrix option should be set as soon as it is created with `GrB_Matrix_new`, so the internal transformation takes less time.

If an error occurs, `GrB_error(&err,A)` returns details about the error.

8.10 GxB_Desc_set: set a GrB_Descriptor value

```
GrB_Info GxB_set                // set a parameter in a descriptor
(
    GrB_Descriptor desc,        // descriptor to modify
    const GrB_Desc_Field field, // parameter to change
    ...                         // value to change it to
) ;
```

This usage is similar to `GrB_Descriptor_set`, just with a name that is consistent with the other usages of this generic function. Unlike `GrB_Descriptor_set`, the `field` may also be `GxB_NTHREADS`, `GxB_CHUNK`, `GxB_SORT`, `GxB_COMPRESSION`, or `GxB_IMPORT`. Refer to Sections 6.14.3 and 6.14.4 for details. If an error occurs, `GrB_error(&err,desc)` returns details about the error.

8.11 GxB_Global_Option_get: retrieve a global option

```
GrB_Info GxB_get                    // gets the current global default option
(
    const GxB_Option_Field field,   // option to query
    ...                             // return value of the global option
) ;
```

This usage of `GxB_get` retrieves the value of a global option. The `field` parameter can be one of the following:

`GxB_HYPER_SWITCH`	sparse/hyper setting
`GxB_BITMAP_SWITCH`	bitmap/sparse setting
`GxB_FORMAT`	by row/col setting
`GxB_MODE`	blocking / non-blocking
`GxB_NTHREADS`	default number of threads
`GxB_CHUNK`	default chunk size
`GxB_BURBLE`	burble setting
`GxB_PRINTF`	printf function
`GxB_FLUSH`	flush function
`GxB_MEMORY_POOL`	memory pool control
`GxB_PRINT_1BASED`	for printing matrices/vectors
`GxB_LIBRARY_NAME`	the string `"SuiteSparse:GraphBLAS"`
`GxB_LIBRARY_VERSION`	`int` array of size 3
`GxB_LIBRARY_DATE`	date of release
`GxB_LIBRARY_ABOUT`	author, copyright
`GxB_LIBRARY_LICENSE`	license for the library
`GxB_LIBRARY_COMPILE_DATE`	date of compilation
`GxB_LIBRARY_COMPILE_TIME`	time of compilation
`GxB_LIBRARY_URL`	URL of the library
`GxB_API_VERSION`	GraphBLAS C API Specification Version
`GxB_API_DATE`	date of the C API Spec.
`GxB_API_ABOUT`	about of the C API Spec.
`GxB_API_URL`	URL of the specification

For example:

```
double h ;
GxB_get (GxB_HYPER_SWITCH, &h) ;
printf ("hyper_switch = %g for all new matrices\n", h) ;

double b [GxB_BITMAP_SWITCH] ;
GxB_get (GxB_BITMAP_SWITCH, b) ;
for (int k = 0 ; k < GxB_NBITMAP_SWITCH ; k++)
{
```

```
    printf ("bitmap_switch [%d] = %g ", k, b [k]) ;
    if (k == 0)
    {
        printf ("for vectors and matrices with 1 row or column\n") ;
    }
    else if (k == GxB_NBITMAP_SWITCH - 1)
    {
        printf ("for matrices with min dimension > %d\n", 1 << (k-1)) ;
    }
    else
    {
        printf ("for matrices with min dimension %d to %d\n",
            (1 << (k-1)) + 1, 1 << k) ;
    }
}

GxB_Format_Value s ;
GxB_get (GxB_FORMAT, &s) ;
if (s == GxB_BY_COL) printf ("all new matrices are stored by column\n") ;
else printf ("all new matrices are stored by row\n") ;

GrB_mode mode ;
GxB_get (GxB_MODE, &mode) ;
if (mode == GrB_BLOCKING) printf ("GrB_init(GrB_BLOCKING) was called.\n") ;
else printf ("GrB_init(GrB_NONBLOCKING) was called.\n") ;

int nthreads_max ;
GxB_get (GxB_NTHREADS, &nthreads_max) ;
printf ("max # of threads to use: %d\n", nthreads_max) ;

double chunk ;
GxB_get (GxB_CHUNK, &chunk) ;
printf ("chunk size: %g\n", chunk) ;

int64_t free_pool_limit [64] ;
GxB_get (GxB_MEMORY_POOL, free_pool_limit) ;
for (int k = 0 ; k < 64 ; k++)
    printf ("pool %d: limit %ld\n", free_pool_limit [k]) ;

char *name ;
int ver [3] ;
GxB_get (GxB_LIBRARY_NAME, &name) ;
GxB_get (GxB_LIBRARY_VERSION, ver) ;
printf ("Library %s, version %d.%d.%d\n", name, ver [0], ver [1], ver [2]) ;
```

8.12 GxB_Matrix_Option_get: retrieve a matrix option

```
GrB_Info GxB_get                    // gets the current option of a matrix
(
    GrB_Matrix A,                   // matrix to query
    GxB_Option_Field field,         // option to query
    ...                             // return value of the matrix option
) ;
```

This usage of `GxB_get` retrieves the value of a matrix option. The `field` parameter can be `GxB_HYPER_SWITCH`, `GxB_BITMAP_SWITCH`, `GxB_SPARSITY_CONTROL`, `GxB_SPARSITY_STATUS`, or `GxB_FORMAT`. For example:

```
double h, b  ;
int sparsity, scontrol ;
GxB_get (A, GxB_SPARSITY_STATUS, &sparsity) ;
GxB_get (A, GxB_HYPER_SWITCH, &h) ;
printf ("matrix A has hyper_switch = %g\n", h) ;
GxB_get (A, GxB_BITMAP_SWITCH, &b) ;
printf ("matrix A has bitmap_switch = %g\n", b) ;
switch (sparsity)
{
    case GxB_HYPERSPARSE: printf ("matrix A is hypersparse\n") ; break ;
    case GxB_SPARSE:      printf ("matrix A is sparse\n"     ) ; break ;
    case GxB_BITMAP:      printf ("matrix A is bitmap\n"     ) ; break ;
    case GxB_FULL:        printf ("matrix A is full\n"       ) ; break ;
}
GxB_Format_Value s ;
GxB_get (A, GxB_FORMAT, &s) ;
printf ("matrix A is stored by %s\n", (s == GxB_BY_COL) ? "col" : "row") ;
GxB_get (A, GxB_SPARSITY_CONTROL, &scontrol) ;
if (scontrol & GxB_HYPERSPARSE) printf ("A may become hypersparse\n") ;
if (scontrol & GxB_SPARSE     ) printf ("A may become sparse\n") ;
if (scontrol & GxB_BITMAP     ) printf ("A may become bitmap\n") ;
if (scontrol & GxB_FULL       ) printf ("A may become full\n") ;
```

8.13 GxB_Desc_get: retrieve a GrB_Descriptor value

```
GrB_Info GxB_get                // get a parameter from a descriptor
(
    GrB_Descriptor desc,        // descriptor to query; NULL means defaults
    GrB_Desc_Field field,       // parameter to query
    ...                         // value of the parameter
) ;
```

This usage is the same as `GxB_Desc_get`. The `field` parameter can be `GrB_OUTP`, `GrB_MASK`, `GrB_INP0`, `GrB_INP1`, `GxB_AxB_METHOD`, `GxB_NTHREADS`, `GxB_CHUNK`, `GxB_SORT`, `GxB_COMPRESSION`, or `GxB_IMPORT`. Refer to Section 6.14.5 for details.

8.14 Summary of usage of GxB_set and GxB_get

The different usages of `GxB_set` and `GxB_get` are summarized below.
To set/get the global options:

```
GxB_set (GxB_HYPER_SWITCH, double h) ;
GxB_set (GxB_HYPER_SWITCH, GxB_ALWAYS_HYPER) ;
GxB_set (GxB_HYPER_SWITCH, GxB_NEVER_HYPER) ;
GxB_get (GxB_HYPER_SWITCH, double *h) ;
double b [GxB_NBITMAP_SWITCH] ;
GxB_set (GxB_BITMAP_SWITCH, b) ;
GxB_set (GxB_BITMAP_SWITCH, NULL) ;     // set defaults
GxB_get (GxB_BITMAP_SWITCH, b) ;
GxB_set (GxB_FORMAT, GxB_BY_ROW) ;
GxB_set (GxB_FORMAT, GxB_BY_COL) ;
GxB_get (GxB_FORMAT, GxB_Format_Value *s) ;
GxB_set (GxB_NTHREADS, int nthreads_max) ;
GxB_get (GxB_NTHREADS, int *nthreads_max) ;
GxB_set (GxB_CHUNK, double chunk) ;
GxB_get (GxB_CHUNK, double *chunk) ;
GxB_set (GxB_BURBLE, bool burble) ;
GxB_get (GxB_BURBLE, bool *burble) ;
GxB_set (GxB_PRINTF, void *printf_function) ;
GxB_get (GxB_PRINTF, void **printf_function) ;
GxB_set (GxB_FLUSH, void *flush_function) ;
GxB_get (GxB_FLUSH, void **flush_function) ;
int64_t free_pool_limit [64] ;
GxB_set (GxB_MEMORY_POOL, free_pool_limit) ;
GxB_set (GxB_MEMORY_POOL, NULL) ;     // set defaults
GxB_get (GxB_MEMORY_POOL, free_pool_limit) ;
GxB_set (GxB_PRINT_1BASED, bool onebased) ;
GxB_get (GxB_PRINT_1BASED, bool *onebased) ;
```

To get global options that can be queried but not modified:

```
GxB_get (GxB_MODE,                 GrB_Mode *mode) ;
GxB_get (GxB_LIBRARY_NAME,         char **) ;
GxB_get (GxB_LIBRARY_VERSION,      int *) ;
GxB_get (GxB_LIBRARY_DATE,         char **) ;
GxB_get (GxB_LIBRARY_ABOUT,        char **) ;
GxB_get (GxB_LIBRARY_LICENSE,      char **) ;
GxB_get (GxB_LIBRARY_COMPILE_DATE, char **) ;
GxB_get (GxB_LIBRARY_COMPILE_TIME, char **) ;
GxB_get (GxB_LIBRARY_URL,          char **) ;
GxB_get (GxB_API_VERSION,          int *) ;
GxB_get (GxB_API_DATE,             char **) ;
GxB_get (GxB_API_ABOUT,            char **) ;
GxB_get (GxB_API_URL,              char **) ;
```

To set/get a matrix option or status

```
GxB_set (GrB_Matrix A, GxB_HYPER_SWITCH, double h) ;
GxB_set (GrB_Matrix A, GxB_HYPER_SWITCH, GxB_ALWAYS_HYPER) ;
GxB_set (GrB_Matrix A, GxB_HYPER_SWITCH, GxB_NEVER_HYPER) ;
GxB_get (GrB_Matrix A, GxB_HYPER_SWITCH, double *h) ;
GxB_set (GrB_Matrix A, GxB_BITMAP_SWITCH, double b) ;
GxB_get (GrB_Matrix A, GxB_BITMAP_SWITCH, double *b) ;
GxB_set (GrB_Matrix A, GxB_FORMAT, GxB_BY_ROW) ;
GxB_set (GrB_Matrix A, GxB_FORMAT, GxB_BY_COL) ;
GxB_get (GrB_Matrix A, GxB_FORMAT, GxB_Format_Value *s) ;
GxB_set (GrB_Matrix A, GxB_SPARSITY_CONTROL, GxB_AUTO_SPARSITY) ;
GxB_set (GrB_Matrix A, GxB_SPARSITY_CONTROL, scontrol) ;
GxB_get (GrB_Matrix A, GxB_SPARSITY_CONTROL, int *scontrol) ;
GxB_get (GrB_Matrix A, GxB_SPARSITY_STATUS, int *sparsity) ;
```

To set/get a vector option or status:

```
GxB_set (GrB_Vector v, GxB_BITMAP_SWITCH, double b) ;
GxB_get (GrB_Vector v, GxB_BITMAP_SWITCH, double *b) ;
GxB_set (GrB_Vector v, GxB_FORMAT, GxB_BY_ROW) ;
GxB_set (GrB_Vector v, GxB_FORMAT, GxB_BY_COL) ;
GxB_get (GrB_Vector v, GxB_FORMAT, GxB_Format_Value *s) ;
GxB_set (GrB_Vector v, GxB_SPARSITY_CONTROL, GxB_AUTO_SPARSITY) ;
GxB_set (GrB_Vector v, GxB_SPARSITY_CONTROL, scontrol) ;
GxB_get (GrB_Vector v, GxB_SPARSITY_CONTROL, int *scontrol) ;
GxB_get (GrB_Vector v, GxB_SPARSITY_STATUS, int *sparsity) ;
```

To set/get a descriptor field:

```
GxB_set (GrB_Descriptor d, GrB_OUTP, GxB_DEFAULT) ;
GxB_set (GrB_Descriptor d, GrB_OUTP, GrB_REPLACE) ;
GxB_get (GrB_Descriptor d, GrB_OUTP, GrB_Desc_Value *v) ;
GxB_set (GrB_Descriptor d, GrB_MASK, GxB_DEFAULT) ;
GxB_set (GrB_Descriptor d, GrB_MASK, GrB_COMP) ;
GxB_set (GrB_Descriptor d, GrB_MASK, GrB_STRUCTURE) ;
GxB_set (GrB_Descriptor d, GrB_MASK, GrB_COMP+GrB_STRUCTURE) ;
GxB_get (GrB_Descriptor d, GrB_MASK, GrB_Desc_Value *v) ;
GxB_set (GrB_Descriptor d, GrB_INP0, GxB_DEFAULT) ;
GxB_set (GrB_Descriptor d, GrB_INP0, GrB_TRAN) ;
GxB_get (GrB_Descriptor d, GrB_INP0, GrB_Desc_Value *v) ;
GxB_set (GrB_Descriptor d, GrB_INP1, GxB_DEFAULT) ;
GxB_set (GrB_Descriptor d, GrB_INP1, GrB_TRAN) ;
GxB_get (GrB_Descriptor d, GrB_INP1, GrB_Desc_Value *v) ;
GxB_set (GrB_Descriptor d, GxB_AxB_METHOD, GxB_DEFAULT) ;
GxB_set (GrB_Descriptor d, GxB_AxB_METHOD, GxB_AxB_GUSTAVSON) ;
GxB_set (GrB_Descriptor d, GxB_AxB_METHOD, GxB_AxB_HASH) ;
GxB_set (GrB_Descriptor d, GxB_AxB_METHOD, GxB_AxB_SAXPY) ;
GxB_set (GrB_Descriptor d, GxB_AxB_METHOD, GxB_AxB_DOT) ;
GxB_get (GrB_Descriptor d, GrB_AxB_METHOD, GrB_Desc_Value *v) ;
GxB_set (GrB_Descriptor d, GxB_NTHREADS, int nthreads) ;
GxB_get (GrB_Descriptor d, GxB_NTHREADS, int *nthreads) ;
GxB_set (GrB_Descriptor d, GxB_CHUNK, double chunk) ;
GxB_get (GrB_Descriptor d, GxB_CHUNK, double *chunk) ;
GxB_set (GrB_Descriptor d, GxB_SORT, sort) ;
GxB_get (GrB_Descriptor d, GxB_SORT, int *sort) ;
GxB_set (GrB_Descriptor d, GxB_COMPRESSION, GxB_FAST_IMPORT) ;
GxB_set (GrB_Descriptor d, GxB_COMPRESSION, GxB_SECURE_IMPORT) ;
GxB_get (GrB_Descriptor d, GxB_COMPRESSION, GrB_Desc_Value *method) ;
GxB_set (GrB_Descriptor d, GxB_IMPORT, int method) ;
GxB_get (GrB_Descriptor d, GxB_IMPORT, int *method) ;
```

9 SuiteSparse:GraphBLAS Colon and Index Notation

Octave/MATLAB uses a colon notation to index into matrices, such as `C=A(2:4,3:8)`, which extracts `C` as 3-by-6 submatrix from `A`, from rows 2 through 4 and columns 3 to 8 of the matrix `A`. A single colon is used to denote all rows, `C=A(:,9)`, or all columns, `C=A(12,:)`, which refers to the 9th column and 12th row of `A`, respectively. An arbitrary integer list can be given as well, such as the Octave/MATLAB statements:

```
I = [2 1 4] ;
J = [3 5] ;
C = A (I,J) ;
```

which creates the 3-by-2 matrix `C` as follows:

$$C = \begin{bmatrix} a_{2,3} & a_{2,5} \\ a_{1,3} & a_{1,5} \\ a_{4,3} & a_{4,5} \end{bmatrix}$$

The GraphBLAS API can do the equivalent of `C=A(I,J)`, `C=A(:,J)`, `C=A(I,:)`, and `C=A(:,:)`, by passing a parameter `const GrB_Index *I` as either an array of size `ni`, or as the special value `GrB_ALL`, which corresponds to the stand-alone colon `C=A(:,J)`, and the same can be done for `J`.. To compute `C=A(2:4,3:8)` in GraphBLAS requires the user application to create two explicit integer arrays `I` and `J` of size 3 and 5, respectively, and then fill them with the explicit values `[2,3,4]` and `[3,4,5,6,7,8]`. This works well if the lists are small, or if the matrix has more entries than rows or columns.

However, particularly with hypersparse matrices, the size of the explicit arrays `I` and `J` can vastly exceed the number of entries in the matrix. When using its hypersparse format, SuiteSparse:GraphBLAS allows the user application to create a `GrB_Matrix` with dimensions up to 2^{60}, with no memory constraints. The only constraint on memory usage in a hypersparse matrix is the number of entries in the matrix.

For example, creating a n-by-n matrix `A` of type `GrB_FP64` with $n = 2^{60}$ and one million entries is trivial to do in Version 2.1 (and later) of SuiteSparse:GraphBLAS, taking at most 24MB of space. SuiteSparse:GraphBLAS Version 2.1 (or later) could do this on an old smartphone. However, using just the pure GraphBLAS API, constructing `C=A(0:(n/2),0:(n/2))` in SuiteSparse Version 2.0 would require the creation of an integer array `I` of size 2^{59}, containing the sequence 0, 1, 2, 3, ..., requiring about 4 ExaBytes of memory (4 million terabytes). This is roughly 1000 times larger than the memory size of the world's largest computer in 2018.

SuiteSparse:GraphBLAS Version 2.1 and later extends the GraphBLAS API with a full implementation of the MATLAB colon notation for integers, `I=begin:inc:end`. This extension allows the construction of the matrix `C=A(0:(n/2),0:(n/2))` in this example, with dimension 2^{59}, probably taking just milliseconds on an old smartphone.

The `GrB_extract`, `GrB_assign`, and `GxB_subassign` operations (described in the Section 10) each have parameters that define a list of integer indices, using two parameters:

```
const GrB_Index *I ;    // an array, or a special value GrB_ALL
GrB_Index ni ;          // the size of I, or a special value
```

These two parameters define five kinds of index lists, which can be used to specify either an explicit or implicit list of row indices and/or column indices. The length of the list of indices is denoted `|I|`. This discussion applies equally to the row indices `I` and the column indices `J`. The five kinds are listed below.

1. An explicit list of indices, such as `I = [2 1 4 7 2]` in MATLAB notation, is handled by passing in `I` as a pointer to an array of size 5, and passing `ni=5` as the size of the list. The length of the explicit list is `ni=|I|`. Duplicates may appear, except that for some uses of `GrB_assign` and `GxB_subassign`, duplicates lead to undefined behavior according to the GraphBLAS C API Specification. SuiteSparse:GraphBLAS specifies how duplicates are handled in all cases, as an addition to the specification. See Section 10.10 for details.

2. To specify all rows of a matrix, use `I = GrB_ALL`. The parameter `ni` is ignored. This is equivalent to `C=A(:,J)` in MATLAB. In GraphBLAS, this is the sequence `0:(m-1)` if `A` has `m` rows, with length `|I|=m`. If `J` is used the columns of an `m`-by-`n` matrix, then `J=GrB_ALL` refers to all columns, and is the sequence `0:(n-1)`, of length `|J|=n`.

   SPEC: If `I` or `J` are `GrB_ALL`, the specification requires that `ni` be passed in as `m` (the number of rows) and `nj` be passed in as `n`. Any other value is an error. SuiteSparse:GraphBLAS ignores these scalar inputs and treats them as if they are equal to their only possible correct value.

3. To specify a contiguous range of indices, such as `I=10:20` in MATLAB, the array `I` has size 2, and `ni` is passed to SuiteSparse:GraphBLAS as the special value `ni = GxB_RANGE`. The beginning index is `I[GxB_BEGIN]` and the ending index is `I[GxB_END]`. Both values must be non-negative since `GrB_Index` is an unsigned integer (`uint64_t`). The value of `I[GxB_INC]` is ignored.

   ```
   // to specify I = 10:20
   GrB_Index I [2], ni = GxB_RANGE ;
   I [GxB_BEGIN] = 10 ;      // the start of the sequence
   I [GxB_END  ] = 20 ;      // the end of the sequence
   ```

   Let $b =$ `I[GxB_BEGIN]`, let $e =$ `I[GxB_END]`, The sequence has length zero if $b > e$; otherwise the length is $|I| = (e - b) + 1$.

4. To specify a strided range of indices with a non-negative stride, such as `I=3:2:10`, the array `I` has size 3, and `ni` has the special value `GxB_STRIDE`. This is the sequence 3, 5, 7, 9, of length 4. Note that 10 does not appear in the list. The end point need not appear if the increment goes past it.

```
// to specify I = 3:2:10
GrB_Index I [3], ni = GxB_STRIDE ;
I [GxB_BEGIN ] = 3 ;      // the start of the sequence
I [GxB_INC   ] = 2 ;      // the increment
I [GxB_END   ] = 10 ;     // the end of the sequence
```

The `GxB_STRIDE` sequence is the same as the `List` generated by the following for loop:

```
int64_t k = 0 ;
GrB_Index *List = (a pointer to an array of large enough size)
for (int64_t i = I [GxB_BEGIN] ; i <= I [GxB_END] ; i += I [GxB_INC])
{
    // i is the kth entry in the sequence
    List [k++] = i ;
}
```

Then passing the explicit array `List` and its length `ni=k` has the same effect as passing in the array `I` of size 3, with `ni=GxB_STRIDE`. The latter is simply much faster to produce, and much more efficient for SuiteSparse:GraphBLAS to process.

Let $b =$ `I[GxB_BEGIN]`, let $e =$ `I[GxB_END]`, and let $\Delta =$ `I[GxB_INC]`. The sequence has length zero if $b > e$ or $\Delta = 0$. Otherwise, the length of the sequence is

$$|I| = \left\lfloor \frac{e - b}{\Delta} \right\rfloor + 1$$

5. In MATLAB notation, if the stride is negative, the sequence is decreasing. For example, `10:-2:1` is the sequence 10, 8, 6, 4, 2, in that order. In SuiteSparse:GraphBLAS, use `ni = GxB_BACKWARDS`, with an array `I` of size 3. The following example specifies defines the equivalent of the MATLAB expression `10:-2:1` in SuiteSparse:GraphBLAS:

```
// to specify I = 10:-2:1
GrB_Index I [3], ni = GxB_BACKWARDS ;
I [GxB_BEGIN ] = 10 ;     // the start of the sequence
I [GxB_INC   ] = 2 ;      // the magnitude of the increment
I [GxB_END   ] = 1 ;      // the end of the sequence
```

The value -2 cannot be assigned to the `GrB_Index` array `I`, since that is an unsigned type. The signed increment is represented instead with the special value `ni = GxB_BACKWARDS`. The `GxB_BACKWARDS` sequence is the same as generated by the following for loop:

```
int64_t k = 0 ;
GrB_Index *List = (a pointer to an array of large enough size)
for (int64_t i = I [GxB_BEGIN] ; i >= I [GxB_END] ; i -= I [GxB_INC])
{
    // i is the kth entry in the sequence
    List [k++] = i ;
}
```

Let b = `I[GxB_BEGIN]`, let e = `I[GxB_END]`, and let Δ = `I[GxB_INC]` (note that Δ is not negative). The sequence has length zero if $b < e$ or $\Delta = 0$. Otherwise, the length of the sequence is

$$|I| = \left\lfloor \frac{b-e}{\Delta} \right\rfloor + 1$$

Since `GrB_Index` is an unsigned integer, all three values `I[GxB_BEGIN]`, `I[GxB_INC]`, and `I[GxB_END]` must be non-negative.

Just as in MATLAB, it is valid to specify an empty sequence of length zero. For example, `I = 5:3` has length zero in MATLAB and the same is true for a `GxB_RANGE` sequence in SuiteSparse:GraphBLAS, with `I[GxB_BEGIN]=5` and `I[GxB_END]=3`. This has the same effect as array `I` with `ni=0`.

10 GraphBLAS Operations

The next sections define each of the GraphBLAS operations, also listed in the table below.

`GrB_mxm`	matrix-matrix multiply	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$
`GrB_vxm`	vector-matrix multiply	$\mathbf{w}^\mathsf{T}\langle\mathbf{m}^\mathsf{T}\rangle = \mathbf{w}^\mathsf{T} \odot \mathbf{u}^\mathsf{T}\mathbf{A}$
`GrB_mxv`	matrix-vector multiply	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{Au}$
`GrB_eWiseMult`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \otimes \mathbf{B})$
	set intersection	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$
`GrB_eWiseAdd`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$
	set union	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`GxB_eWiseUnion`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$
	set union	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`GrB_extract`	extract submatrix	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}(\mathbf{I}, \mathbf{J})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{u}(\mathbf{i})$
`GxB_subassign`	assign submatrix,	$\mathbf{C}(\mathbf{I}, \mathbf{J})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{I}, \mathbf{J}) \odot \mathbf{A}$
	with submask for $\mathbf{C}(\mathbf{I}, \mathbf{J})$	$\mathbf{w}(\mathbf{i})\langle\mathbf{m}\rangle = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GrB_assign`	assign submatrix	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I}, \mathbf{J}) = \mathbf{C}(\mathbf{I}, \mathbf{J}) \odot \mathbf{A}$
	with submask for $\mathbf{C}$	$\mathbf{w}\langle\mathbf{m}\rangle(\mathbf{i}) = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GrB_apply`	apply unary operator	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u})$
	apply binary operator	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(x, \mathbf{A})$
		$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, y)$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(x, \mathbf{x})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, y)$
	apply index-unary op	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, i, j, k)$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, i, 0, k)$
`GrB_select`	select entries	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \text{select}(\mathbf{A}, i, j, k)$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \text{select}(\mathbf{u}, i, 0, k)$
`GrB_reduce`	reduce to vector	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$
	reduce to scalar	$s = s \odot [\oplus_{ij} \mathbf{A}(I, J)]$
`GrB_transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}^\mathsf{T}$
`GrB_kronecker`	Kronecker product	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \text{kron}(\mathbf{A}, \mathbf{B})$

If an error occurs, `GrB_error(&err,C)` or `GrB_error(&err,w)` returns details about the error, for operations that return a modified matrix `C` or vector `w`. The only operation that cannot return an error string is reduction to a scalar with `GrB_reduce`.

10.1 GrB_mxm: matrix-matrix multiply

```
GrB_Info GrB_mxm                    // C<Mask> = accum (C, A*B)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_Semiring semiring,    // defines '+' and '*' for A*B
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Matrix B,             // second input: matrix B
    const GrB_Descriptor desc       // descriptor for C, Mask, A, and B
) ;
```

`GrB_mxm` multiplies two sparse matrices `A` and `B` using the `semiring`. The input matrices `A` and `B` may be transposed according to the descriptor, `desc` (which may be `NULL`) and then typecasted to match the multiply operator of the `semiring`. Next, `T=A*B` is computed on the `semiring`, precisely defined in the `GB_spec_mxm.m` script in `GraphBLAS/Test`. The actual algorithm exploits sparsity and does not take $O(n^3)$ time, but it computes the following:

```
[m s] = size (A.matrix) ;
[s n] = size (B.matrix) ;
T.matrix  = zeros (m, n, multiply.ztype) ;
T.pattern = zeros (m, n, 'logical') ;
T.matrix (:,:) = identity ;             % the identity of the semiring's monoid
T.class = multiply.ztype ;              % the ztype of the semiring's multiply op
A = cast (A.matrix, multiply.xtype) ;   % the xtype of the semiring's multiply op
B = cast (B.matrix, multiply.ytype) ;   % the ytype of the semiring's multiply op
for j = 1:n
    for i = 1:m
        for k = 1:s
            % T (i,j) += A (i,k) * B (k,j), using the semiring
            if (A.pattern (i,k) && B.pattern (k,j))
                z = multiply (A (i,k), B (k,j)) ;
                T.matrix  (i,j) = add (T.matrix (i,j),  z) ;
                T.pattern (i,j) = true ;
            end
        end
    end
end
```

Finally, `T` is typecasted into the type of `C`, and the results are written back into `C` via the `accum` and `Mask`, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$. The latter step is reflected in the MATLAB function `GB_spec_accum_mask.m`, discussed in Section 2.3.

Performance considerations: Suppose all matrices are in `GxB_BY_COL` format, and `B` is extremely sparse but `A` is not as sparse. Then computing `C=A*B` is very fast, and much faster than when `A` is extremely sparse. For example, if `A` is square and `B` is a column vector that is all nonzero except for one entry `B(j,0)=1`, then `C=A*B` is the same as extracting column `A(:,j)`. This is very fast if `A` is stored by column but slow if `A` is stored by row. If `A` is a sparse row with a single entry `A(0,i)=1`, then `C=A*B` is the same as extracting row `B(i,:)`. This is fast if `B` is stored by row but slow if `B` is stored by column.

If the user application needs to repeatedly extract rows and columns from a matrix, whether by matrix multiplication or by `GrB_extract`, then keep two copies: one stored by row, and other by column, and use the copy that results in the fastest computation.

By default, `GrB_mxm`, `GrB_mxv`, `GrB_vxm`, and `GrB_reduce` (to vector) can return their result in a jumbled state, with the sort left pending. It can sometimes be faster for these methods to do the sort as they compute their result. Use the `GxB_SORT` descriptor setting to select this option. Refer to Section 6.14 for details.

10.2 GrB_vxm: vector-matrix multiply

```
GrB_Info GrB_vxm                    // w'<mask> = accum (w, u'*A)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_Semiring semiring,    // defines '+' and '*' for u'*A
    const GrB_Vector u,             // first input:  vector u
    const GrB_Matrix A,             // second input: matrix A
    const GrB_Descriptor desc       // descriptor for w, mask, and A
) ;
```

`GrB_vxm` multiplies a row vector `u'` times a matrix `A`. The matrix `A` may be first transposed according to `desc` (as the second input, `GrB_INP1`); the column vector `u` is never transposed via the descriptor. The inputs `u` and `A` are typecasted to match the `xtype` and `ytype` inputs, respectively, of the multiply operator of the `semiring`. Next, an intermediate column vector `t=A'*u` is computed on the `semiring` using the same method as `GrB_mxm`. Finally, the column vector `t` is typecasted from the `ztype` of the multiply operator of the `semiring` into the type of `w`, and the results are written back into `w` using the optional accumulator `accum` and `mask`.

The last step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

Performance considerations: If the `GxB_FORMAT` of `A` is `GxB_BY_ROW`, and the default descriptor is used (`A` is not transposed), then `GrB_vxm` is faster than than `GrB_mxv` with its default descriptor, when the vector `u` is very sparse. However, if the `GxB_FORMAT` of `A` is `GxB_BY_COL`, then `GrB_mxv` with its default descriptor is faster than `GrB_vxm` with its default descriptor, when the vector `u` is very sparse. Using the non-default `GrB_TRAN` descriptor for `A` makes the `GrB_vxm` operation equivalent to `GrB_mxv` with its default descriptor (with the operands reversed in the multiplier, as well). The reverse is true as well; `GrB_mxv` with `GrB_TRAN` is the same as `GrB_vxm` with a default descriptor.

10.3 GrB_mxv: matrix-vector multiply

```
GrB_Info GrB_mxv                    // w<mask> = accum (w, A*u)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_Semiring semiring,    // defines '+' and '*' for A*B
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Vector u,             // second input: vector u
    const GrB_Descriptor desc       // descriptor for w, mask, and A
) ;
```

`GrB_mxv` multiplies a matrix `A` times a column vector `u`. The matrix `A` may be first transposed according to `desc` (as the first input); the column vector `u` is never transposed via the descriptor. The inputs `A` and `u` are typecasted to match the `xtype` and `ytype` inputs, respectively, of the multiply operator of the `semiring`. Next, an intermediate column vector `t=A*u` is computed on the `semiring` using the same method as `GrB_mxm`. Finally, the column vector `t` is typecasted from the `ztype` of the multiply operator of the `semiring` into the type of `w`, and the results are written back into `w` using the optional accumulator `accum` and `mask`.

The last step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

Performance considerations: Refer to the discussion of `GrB_vxm`. In SuiteSparse:GraphBLAS, `GrB_mxv` is very efficient when `u` is sparse or dense, when the default descriptor is used, and when the matrix is `GxB_BY_COL`. When `u` is very sparse and `GrB_INP0` is set to its non-default `GrB_TRAN`, then this method is not efficient if the matrix is in `GxB_BY_COL` format. If an application needs to perform `A'*u` repeatedly where `u` is very sparse, then use the `GxB_BY_ROW` format for `A` instead.

10.4 GrB_eWiseMult: element-wise operations, set intersection

Element-wise "multiplication" is shorthand for applying a binary operator element-wise on two matrices or vectors `A` and `B`, for all entries that appear in the set intersection of the patterns of `A` and `B`. This is like `A.*B` for two sparse matrices in MATLAB, except that in GraphBLAS any binary operator can be used, not just multiplication.

The pattern of the result of the element-wise "multiplication" is exactly this set intersection. Entries in `A` but not `B`, or visa versa, do not appear in the result.

Let $\otimes$ denote the binary operator to be used. The computation $\mathbf{T} = \mathbf{A} \otimes \mathbf{B}$ is given below. Entries not in the intersection of $\mathbf{A}$ and $\mathbf{B}$ do not appear in the pattern of $\mathbf{T}$. That is:

for all entries (i, j) in $\mathbf{A} \cap \mathbf{B}$
$t_{ij} = a_{ij} \otimes b_{ij}$

Depending on what kind of operator is used and what the implicit value is assumed to be, this can give the Hadamard product. This is the case for `A.*B` in MATLAB since the implicit value is zero. However, computing a Hadamard product is not necessarily the goal of the `eWiseMult` operation. It simply applies any binary operator, built-in or user-defined, to the set intersection of `A` and `B`, and discards any entry outside this intersection. Its usefulness in a user's application does not depend upon it computing a Hadamard product in all cases. The operator need not be associative, commutative, nor have any particular property except for type compatibility with `A` and `B`, and the output matrix `C`.

The generic name for this operation is `GrB_eWiseMult`, which can be used for both matrices and vectors.

10.4.1 GrB_Vector_eWiseMult: **element-wise vector multiply**

```
GrB_Info GrB_eWiseMult              // w<mask> = accum (w, u.*v)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const <operator> multiply,      // defines '.*' for t=u.*v
    const GrB_Vector u,             // first input:  vector u
    const GrB_Vector v,             // second input: vector v
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_eWiseMult` computes the element-wise "multiplication" of two vectors `u` and `v`, element-wise using any binary operator (not just times). The vectors are not transposed via the descriptor. The vectors `u` and `v` are first typecasted into the first and second inputs of the `multiply` operator. Next, a column vector `t` is computed, denoted $\mathbf{t} = \mathbf{u} \otimes \mathbf{v}$. The pattern of `t` is the set intersection of `u` and `v`. The result `t` has the type of the output `ztype` of the `multiply` operator.

The `operator` is typically a `GrB_BinaryOp`, but the method is type-generic for this parameter. If given a monoid (`GrB_Monoid`), the additive operator of the monoid is used as the `multiply` binary operator. If given a semiring (`GrB_Semiring`), the multiply operator of the semiring is used as the `multiply` binary operator.

The next and final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices. Note for all GraphBLAS operations, including this one, the accumulator $\mathbf{w} \odot \mathbf{t}$ is always applied in a set union manner, even though $\mathbf{t} = \mathbf{u} \otimes \mathbf{v}$ for this operation is applied in a set intersection manner.

10.4.2 GrB_Matrix_eWiseMult: element-wise matrix multiply

```
GrB_Info GrB_eWiseMult              // C<Mask> = accum (C, A.*B)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const <operator> multiply,      // defines '.*' for T=A.*B
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Matrix B,             // second input: matrix B
    const GrB_Descriptor desc       // descriptor for C, Mask, A, and B
) ;
```

`GrB_Matrix_eWiseMult` computes the element-wise "multiplication" of two matrices `A` and `B`, element-wise using any binary operator (not just times). The input matrices may be transposed first, according to the descriptor `desc`. They are then typecasted into the first and second inputs of the `multiply` operator. Next, a matrix `T` is computed, denoted $\mathbf{T} = \mathbf{A} \otimes \mathbf{B}$. The pattern of `T` is the set intersection of `A` and `B`. The result `T` has the type of the output `ztype` of the `multiply` operator.

The `multiply` operator is typically a `GrB_BinaryOp`, but the method is type-generic for this parameter. If given a monoid (`GrB_Monoid`), the additive operator of the monoid is used as the `multiply` binary operator. If given a semiring (`GrB_Semiring`), the multiply operator of the semiring is used as the `multiply` binary operator.

The operation can be expressed in MATLAB notation as:

```
[nrows, ncols] = size (A.matrix) ;
T.matrix = zeros (nrows, ncols, multiply.ztype) ;
T.class = multiply.ztype ;
p = A.pattern & B.pattern ;
A = cast (A.matrix (p), multiply.xtype) ;
B = cast (B.matrix (p), multiply.ytype) ;
T.matrix (p) = multiply (A, B) ;
T.pattern = p ;
```

The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3. Note for all GraphBLAS operations, including this one, the accumulator $\mathbf{C} \odot \mathbf{T}$ is always applied in a set union manner, even though $\mathbf{T} = \mathbf{A} \otimes \mathbf{B}$ for this operation is applied in a set intersection manner.

10.5 GrB_eWiseAdd: element-wise operations, set union

Element-wise "addition" is shorthand for applying a binary operator element-wise on two matrices or vectors `A` and `B`, for all entries that appear in the set intersection of the patterns of `A` and `B`. This is like `A+B` for two sparse matrices in MATLAB, except that in GraphBLAS any binary operator can be used, not just addition. The pattern of the result of the element-wise "addition" is the set union of the pattern of `A` and `B`. Entries in neither in `A` nor in `B` do not appear in the result.

Let $\oplus$ denote the binary operator to be used. The computation $\mathbf{T} = \mathbf{A} \oplus \mathbf{B}$ is exactly the same as the computation with accumulator operator as described in Section 2.3. It acts like a sparse matrix addition, except that any operator can be used. The pattern of $\mathbf{A} \oplus \mathbf{B}$ is the set union of the patterns of $\mathbf{A}$ and $\mathbf{B}$, and the operator is applied only on the set intersection of $\mathbf{A}$ and $\mathbf{B}$. Entries not in either the pattern of $\mathbf{A}$ or $\mathbf{B}$ do not appear in the pattern of $\mathbf{T}$. That is:

$$
\begin{array}{l}
\text{for all entries } (i,j) \text{ in } \mathbf{A} \cap \mathbf{B} \\
\qquad t_{ij} = a_{ij} \oplus b_{ij} \\
\text{for all entries } (i,j) \text{ in } \mathbf{A} \setminus \mathbf{B} \\
\qquad t_{ij} = a_{ij} \\
\text{for all entries } (i,j) \text{ in } \mathbf{B} \setminus \mathbf{A} \\
\qquad t_{ij} = b_{ij}
\end{array}
$$

The only difference between element-wise "multiplication" ($\mathbf{T} = \mathbf{A} \otimes \mathbf{B}$) and "addition" ($\mathbf{T} = \mathbf{A} \oplus \mathbf{B}$) is the pattern of the result, and what happens to entries outside the intersection. With $\otimes$ the pattern of $\mathbf{T}$ is the intersection; with $\oplus$ it is the set union. Entries outside the set intersection are dropped for $\otimes$, and kept for $\oplus$; in both cases the operator is only applied to those (and only those) entries in the intersection. Any binary operator can be used interchangeably for either operation.

Element-wise operations do not operate on the implicit values, even implicitly, since the operations make no assumption about the semiring. As a result, the results can be different from MATLAB, which can always assume the implicit value is zero. For example, `C=A-B` is the conventional matrix subtraction in MATLAB. Computing `A-B` in GraphBLAS with `eWiseAdd` will apply the `MINUS` operator to the intersection, entries in `A` but not `B` will be unchanged and appear in `C`, and entries in neither `A` nor `B` do not appear in `C`. For these cases, the results matches the MATLAB `C=A-B`. Entries in `B` but not `A` do appear in `C` but they are not negated; they cannot be subtracted from an implicit value in `A`. This is by design. If conventional matrix subtraction of two sparse matrices is required, and the implicit value is known to be zero, use `GrB_apply` to negate the values in `B`, and then use `eWiseAdd` with the `PLUS` operator, to compute `A+(-B)`.

The generic name for this operation is `GrB_eWiseAdd`, which can be used for both matrices and vectors.

There is another minor difference in two variants of the element-wise functions. If given a `semiring`, the `eWiseAdd` functions use the binary operator of the semiring's monoid, while the `eWiseMult` functions use the multiplicative operator of the semiring.

10.5.1 GrB_Vector_eWiseAdd: element-wise vector addition

```
GrB_Info GrB_eWiseAdd               // w<mask> = accum (w, u+v)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const <operator> add,           // defines '+' for t=u+v
    const GrB_Vector u,             // first input:  vector u
    const GrB_Vector v,             // second input: vector v
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_eWiseAdd` computes the element-wise "addition" of two vectors `u` and `v`, element-wise using any binary operator (not just plus). The vectors are not transposed via the descriptor. Entries in the intersection of `u` and `v` are first typecasted into the first and second inputs of the `add` operator. Next, a column vector `t` is computed, denoted $\mathbf{t} = \mathbf{u} \oplus \mathbf{v}$. The pattern of `t` is the set union of `u` and `v`. The result `t` has the type of the output `ztype` of the `add` operator.

The `add` operator is typically a `GrB_BinaryOp`, but the method is type-generic for this parameter. If given a monoid (`GrB_Monoid`), the additive operator of the monoid is used as the `add` binary operator. If given a semiring (`GrB_Semiring`), the additive operator of the monoid of the semiring is used as the `add` binary operator.

The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

10.5.2 GrB_Matrix_eWiseAdd: element-wise matrix addition

```
GrB_Info GrB_eWiseAdd               // C<Mask> = accum (C, A+B)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const <operator> add,           // defines '+' for T=A+B
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Matrix B,             // second input: matrix B
    const GrB_Descriptor desc       // descriptor for C, Mask, A, and B
) ;
```

`GrB_Matrix_eWiseAdd` computes the element-wise "addition" of two matrices `A` and `B`, element-wise using any binary operator (not just plus). The input matrices may be transposed first, according to the descriptor `desc`. Entries in the intersection then typecasted into the first and second inputs of the `add` operator. Next, a matrix `T` is computed, denoted $\mathbf{T} = \mathbf{A} \oplus \mathbf{B}$. The pattern of `T` is the set union of `A` and `B`. The result `T` has the type of the output `ztype` of the `add` operator.

The `add` operator is typically a `GrB_BinaryOp`, but the method is type-generic for this parameter. If given a monoid (`GrB_Monoid`), the additive operator of the monoid is used as the `add` binary operator. If given a semiring (`GrB_Semiring`), the additive operator of the monoid of the semiring is used as the `add` binary operator.

The operation can be expressed in MATLAB notation as:

```
[nrows, ncols] = size (A.matrix) ;
T.matrix = zeros (nrows, ncols, add.ztype) ;
p = A.pattern & B.pattern ;
A = GB_mex_cast (A.matrix (p), add.xtype) ;
B = GB_mex_cast (B.matrix (p), add.ytype) ;
T.matrix (p) = add (A, B) ;
p =  A.pattern & ~B.pattern ; T.matrix (p) = cast (A.matrix (p), add.ztype) ;
p = ~A.pattern &  B.pattern ; T.matrix (p) = cast (B.matrix (p), add.ztype) ;
T.pattern = A.pattern | B.pattern ;
T.class = add.ztype ;
```

Except for when typecasting is performed, this is identical to how the `accum` operator is applied in Figure 1.

The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

10.6 GxB_eWiseUnion: element-wise operations, set union

`GxB_eWiseUnion` computes a result with the same pattern `GrB_eWiseAdd`, namely, a set union of its two inputs. It differs in how the binary operator is applied.

Let $\oplus$ denote the binary operator to be used. The operator is applied to every entry in $\mathbf{A}$ and $\mathbf{B}$. A pair of scalars, α and β (`alpha` and `beta` in the API, respectively) define the inputs to the operator when entries are present in one matrix but not the other.

for all entries (i, j) in $\mathbf{A} \cap \mathbf{B}$
$t_{ij} = a_{ij} \oplus b_{ij}$
for all entries (i, j) in $\mathbf{A} \setminus \mathbf{B}$
$t_{ij} = a_{ij} \oplus \beta$
for all entries (i, j) in $\mathbf{B} \setminus \mathbf{A}$
$t_{ij} = \alpha \oplus b_{ij}$

`GxB_eWiseUnion` is useful in contexts where `GrB_eWiseAdd` cannot be used because of the typecasting rules of GraphBLAS. In particular, suppose `A` and `B` are matrices with a user-defined type, and suppose `<` is a user-defined operator that compares two entries of this type and returns a Boolean value. Then `C=A<B` can be computed with `GxB_eWiseUnion` but not with `GrB_eWiseAdd`. In the latter, if `A(i,j)` is present but `B(i,j)` is not, then `A(i,j)` must typecasted to the type of `C` (`GrB_BOOL` in this case), and the assigment `C(i,j) = (bool) A(i,j)` would be performed. This is not possible because user-defined types cannot be typecasted to any other type.

Another advantage of `GxB_eWiseUnion` is its performance. For example, the Octave/MATLAB expression `C=A-B` computes `C(i,j)=-B(i,j)` when `A(i,j)` is not present. This cannot be done with a single call `GrB_eWiseAdd`, but it can be done with a single call to `GxB_eWiseUnion`, with the `GrB_MINUS_FP64` operator, and with both `alpha` and `beta` scalars equal to zero. It is possible to compute this result with a temporary matrix, `E=-B`, computed with `GrB_apply` and `GrB_AINV_FP64`, followed by a call to `GrB_eWiseAdd` to compute `C=A+E`, but this is slower than a single call to `GxB_eWiseUnion`, and uses more memory.

10.6.1 GxB_Vector_eWiseUnion: element-wise vector addition

```
GrB_Info GxB_eWiseUnion             // w<mask> = accum (w, u+v)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_BinaryOp add,         // defines '+' for t=u+v
    const GrB_Vector u,             // first input:  vector u
    const GrB_Scalar alpha,
    const GrB_Vector v,             // second input: vector v
    const GrB_Scalar beta,
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

Identical to `GrB_Vector_eWiseAdd` except that two scalars are used to define how to compute the result when entries are present in one of the two input vectors (`u` and `v`), but not the other. Each of the two input scalars, `alpha` and `beta` must contain an entry. When computing the result `t=u+v`, if `u(i)` is present but `v(i)` is not, then `t(i)=u(i)+beta`. Likewise, if `v(i)` is present but `u(i)` is not, then `t(i)=alpha+v(i)`, where `+` denotes the binary operator, `add`.

10.6.2 GxB_Matrix_eWiseUnion: element-wise matrix addition

```
GrB_Info GxB_eWiseUnion             // C<M> = accum (C, A+B)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_BinaryOp add,         // defines '+' for T=A+B
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Scalar alpha,
    const GrB_Matrix B,             // second input: matrix B
    const GrB_Scalar beta,
    const GrB_Descriptor desc       // descriptor for C, M, A, and B
) ;
```

Identical to `GrB_Matrix_eWiseAdd` except that two scalars are used to define how to compute the result when entries are present in one of the two input matrices (`A` and `B`), but not the other. Each of the two input scalars, `alpha` and `beta` must contain an entry. When computing the result `T=A+B`, if `A(i,j)` is present but `B(i,j))` is not, then `T(i,j)=A(i,j)+beta`. Likewise, if `B(i,j)` is present but `A(i,j)` is not, then `T(i,j)=alpha+B(i,j)`, where `+` denotes the binary operator, `add`.

10.7 GrB_extract: submatrix extraction

The `GrB_extract` function is a generic name for three specific functions: `GrB_Vector_extract`, `GrB_Col_extract`, and `GrB_Matrix_extract`. The generic name appears in the function signature, but the specific function name is used when describing what each variation does.

10.7.1 GrB_Vector_extract: extract subvector from vector

```
GrB_Info GrB_extract                // w<mask> = accum (w, u(I))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_Vector u,             // first input:  vector u
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_extract` extracts a subvector from another vector, identical to `t = u (I)` in MATLAB where `I` is an integer vector of row indices. Refer to `GrB_Matrix_extract` for further details; vector extraction is the same as matrix extraction with `n`-by-1 matrices. See Section 9 for a description of `I` and `ni`. The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

10.7.2 GrB_Matrix_extract: extract submatrix from matrix

```
GrB_Info GrB_extract                // C<Mask> = accum (C, A(I,J))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C, Mask, and A
) ;
```

`GrB_Matrix_extract` extracts a submatrix from another matrix, identical to `T = A(I,J)` in MATLAB where `I` and `J` are integer vectors of row and column indices, respectively, except that indices are zero-based in GraphBLAS and one-based in MATLAB. The input matrix `A` may be transposed first, via the descriptor. The type of `T` and `A` are the same. The size of `C` is `|I|`-by-`|J|`. Entries outside `A(I,J)` are not accessed and do not take part in the computation. More precisely, assuming the matrix `A` is not transposed, the matrix `T` is defined as follows:

```
T.matrix  = zeros (ni, nj) ;    % a matrix of size ni-by-nj
T.pattern = false (ni, nj) ;
for i = 1:ni
    for j = 1:nj
        if (A (I(i),J(j)).pattern)
            T (i,j).matrix  = A (I(i),J(j)).matrix ;
            T (i,j).pattern = true ;
        end
    end
end
```

If duplicate indices are present in `I` or `J`, the above method defines the result in `T`. Duplicates result in the same values of `A` being copied into different places in `T`. See Section 9 for a description of the row indices `I` and `ni`, and the column indices `J` and `nj`. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

Performance considerations: If `A` is not transposed via input descriptor: if `|I|` is small, then it is fastest if `A` is `GxB_BY_ROW`; if `|J|` is small, then it is fastest if `A` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

10.7.3 GrB_Col_extract: extract column vector from matrix

```
GrB_Info GrB_extract                // w<mask> = accum (w, A(I,j))
(
    GrB_Vector w,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index j,              // column index
    const GrB_Descriptor desc       // descriptor for w, mask, and A
) ;
```

`GrB_Col_extract` extracts a subvector from a matrix, identical to `t = A (I,j)` in MATLAB where `I` is an integer vector of row indices and where `j` is a single column index. The input matrix `A` may be transposed first, via the descriptor, which results in the extraction of a single row `j` from the matrix `A`, the result of which is a column vector `w`. The type of `t` and `A` are the same. The size of `w` is `|I|`-by-1.

See Section 9 for a description of the row indices `I` and `ni`. The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

Performance considerations: If `A` is not transposed: it is fastest if the format of `A` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

10.8 GxB_subassign: submatrix assignment

The methods described in this section are all variations of the form `C(I,J)=A`, which modifies a submatrix of the matrix `C`. All methods can be used in their generic form with the single name `GxB_subassign`. This is reflected in the prototypes. However, to avoid confusion between the different kinds of assignment, the name of the specific function is used when describing each variation. If the discussion applies to all variations, the simple name `GxB_subassign` is used.

See Section 9 for a description of the row indices `I` and `ni`, and the column indices `J` and `nj`.

`GxB_subassign` is very similar to `GrB_assign`, described in Section 10.9. The two operations are compared and contrasted in Section 10.11. For a discussion of how duplicate indices are handled in `I` and `J`, see Section 10.10.

10.8.1 GxB_Vector_subassign: assign to a subvector

```
GrB_Info GxB_subassign              // w(I)<mask> = accum (w(I),u)
(
    GrB_Vector w,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for w(I), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w(I),t)
    const GrB_Vector u,             // first input:  vector u
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Descriptor desc       // descriptor for w(I) and mask
) ;
```

`GxB_Vector_subassign` operates on a subvector `w(I)` of `w`, modifying it with the vector `u`. The method is identical to `GxB_Matrix_subassign` described in Section 10.8.2, where all matrices have a single column each. The `mask` has the same size as `w(I)` and `u`. The only other difference is that the input `u` in this method is not transposed via the `GrB_INP0` descriptor.

10.8.2 GxB_Matrix_subassign: assign to a submatrix

```
GrB_Info GxB_subassign              // C(I,J)<Mask> = accum (C(I,J),A)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C(I,J), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C(I,J),T)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C(I,J), Mask, and A
) ;
```

`GxB_Matrix_subassign` operates only on a submatrix `S` of `C`, modifying it with the matrix `A`. For this operation, the result is not the entire matrix `C`, but a submatrix `S=C(I,J)` of `C`. The steps taken are as follows, except that $\mathbf{A}$ may be optionally transposed via the `GrB_INP0` descriptor option.

Step	GraphBLAS notation	description
1	$\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$	extract the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix
2	$\mathbf{S}\langle\mathbf{M}\rangle = \mathbf{S} \odot \mathbf{A}$	apply the accumulator/mask to the submatrix $\mathbf{S}$
3	$\mathbf{C}(\mathbf{I}, \mathbf{J}) = \mathbf{S}$	put the submatrix $\mathbf{S}$ back into $\mathbf{C}(\mathbf{I}, \mathbf{J})$

The accumulator/mask step in Step 2 is the same as for all other GraphBLAS operations, described in Section 2.3, except that for `GxB_subassign`, it is applied to just the submatrix $\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$, and thus the `Mask` has the same size as $\mathbf{A}$, $\mathbf{S}$, and $\mathbf{C}(\mathbf{I}, \mathbf{J})$.

The `GxB_subassign` operation is the reverse of matrix extraction:

- For submatrix extraction, `GrB_Matrix_extract`, the submatrix `A(I,J)` appears on the right-hand side of the assignment, `C=A(I,J)`, and entries outside of the submatrix are not accessed and do not take part in the computation.
- For submatrix assignment, `GxB_Matrix_subassign`, the submatrix `C(I,J)` appears on the left-hand-side of the assignment, `C(I,J)=A`, and entries outside of the submatrix are not accessed and do not take part in the computation.

In both methods, the accumulator and mask modify the submatrix of the assignment; they simply differ on which side of the assignment the submatrix resides on. In both cases, if the `Mask` matrix is present it is the same size as the submatrix:

- For submatrix extraction, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}(\mathbf{I}, \mathbf{J})$ is computed, where the submatrix is on the right. The mask $\mathbf{M}$ has the same size as the submatrix $\mathbf{A}(\mathbf{I}, \mathbf{J})$.
- For submatrix assignment, $\mathbf{C}(\mathbf{I}, \mathbf{J})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{I}, \mathbf{J}) \odot \mathbf{A}$ is computed, where the submatrix is on the left. The mask $\mathbf{M}$ has the same size as the submatrix $\mathbf{C}(\mathbf{I}, \mathbf{J})$.

In Step 1, the submatrix `S` is first computed by the `GrB_Matrix_extract` operation, `S=C(I,J)`.

Step 2 accumulates the results $\mathbf{S}\langle\mathbf{M}\rangle = \mathbf{S} \odot \mathbf{T}$, exactly as described in Section 2.3, but operating on the submatrix $\mathbf{S}$, not $\mathbf{C}$, using the optional `Mask` and `accum` operator. The matrix $\mathbf{T}$ is simply $\mathbf{T} = \mathbf{A}$, or $\mathbf{T} = \mathbf{A}^{\mathsf{T}}$ if $\mathbf{A}$ is transposed via the `desc` descriptor, `GrB_INP0`. The `GrB_REPLACE` option in the descriptor clears $\mathbf{S}$ after computing $\mathbf{Z} = \mathbf{T}$ or $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$, not all of $\mathbf{C}$ since this operation can only modify the specified submatrix of $\mathbf{C}$.

Finally, Step 3 writes the result (which is the modified submatrix `S` and not all of `C`) back into the `C` matrix that contains it, via the assignment `C(I,J)=S`, using the reverse operation from the method described for matrix extraction:

```
for i = 1:ni
    for j = 1:nj
        if (S (i,j).pattern)
            C (I(i),J(j)).matrix = S (i,j).matrix ;
            C (I(i),J(j)).pattern = true ;
        end
    end
end
```

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GxB_BY_ROW`; if `|J|` is small, then it is fastest if the format of `C` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

10.8.3 GxB_Col_subassign: assign to a sub-column of a matrix

```
GrB_Info GxB_subassign              // C(I,j)<mask> = accum (C(I,j),u)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(I,j), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(I,j),t)
    const GrB_Vector u,             // input vector
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index j,              // column index
    const GrB_Descriptor desc       // descriptor for C(I,j) and mask
) ;
```

`GxB_Col_subassign` modifies a single sub-column of a matrix `C`. It is the same as `GxB_Matrix_subassign` where the index vector `J[0]=j` is a single column index (and thus `nj=1`), and where all matrices in `GxB_Matrix_subassign` (except `C`) consist of a single column. The `mask` vector has the same size as `u` and the sub-column `C(I,j)`. The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GxB_Matrix_subassign` for further details.

Performance considerations: `GxB_Col_subassign` is much faster than `GxB_Row_subassign` if the format of `C` is `GxB_BY_COL`. `GxB_Row_subassign` is much faster than `GxB_Col_subassign` if the format of `C` is `GxB_BY_ROW`.

10.8.4 GxB_Row_subassign: assign to a sub-row of a matrix

```
GrB_Info GxB_subassign              // C(i,J)<mask'> = accum (C(i,J),u')
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(i,J), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(i,J),t)
    const GrB_Vector u,             // input vector
    const GrB_Index i,              // row index
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C(i,J) and mask
) ;
```

`GxB_Row_subassign` modifies a single sub-row of a matrix `C`. It is the same as `GxB_Matrix_subassign` where the index vector `I[0]=i` is a single row index (and thus `ni=1`), and where all matrices in `GxB_Matrix_subassign` (except `C`) consist of a single row. The `mask` vector has the same size as `u` and the sub-column `C(I,j)`. The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GxB_Matrix_subassign` for further details.

Performance considerations: `GxB_Col_subassign` is much faster than `GxB_Row_subassign` if the format of `C` is `GxB_BY_COL`. `GxB_Row_subassign` is much faster than `GxB_Col_subassign` if the format of `C` is `GxB_BY_ROW`.

10.8.5 GxB_Vector_subassign_<type>: assign a scalar to a subvector

```
GrB_Info GxB_subassign              // w(I)<mask> = accum (w(I),x)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w(I), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w(I),x)
    const <type> x,                 // scalar to assign to w(I)
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Descriptor desc       // descriptor for w(I) and mask
) ;
```

`GxB_Vector_subassign_<type>` assigns a single scalar to an entire subvector of the vector `w`. The operation is exactly like setting a single entry in an `n`-by-1 matrix, `A(I,0) = x`, where the column index for a vector is implicitly `j=0`. For further details of this function, see `GxB_Matrix_subassign_<type>` in Section 10.8.6.

10.8.6 GxB_Matrix_subassign_<type>: assign a scalar to a submatrix

```
GrB_Info GxB_subassign              // C(I,J)<Mask> = accum (C(I,J),x)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C(I,J), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C(I,J),x)
    const <type> x,                 // scalar to assign to C(I,J)
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C(I,J) and Mask
) ;
```

`GxB_Matrix_subassign_<type>` assigns a single scalar to an entire submatrix of `C`, like the *scalar expansion* `C(I,J)=x` in MATLAB. The scalar `x` is implicitly expanded into a matrix `A` of size `ni` by `nj`, with all entries present and equal to `x`, and then the matrix `A` is assigned to `C(I,J)` using the same method as in `GxB_Matrix_subassign`. Refer to that function in Section 10.8.2 for further details. For the accumulation step, the scalar `x` is typecasted directly into the type of `C` when the `accum` operator is not applied to it, or into the `ytype` of the `accum` operator, if `accum` is not NULL, for entries that are already present in `C`.

The `<type> x` notation is otherwise the same as `GrB_Matrix_setElement` (see Section 6.9.11). Any value can be passed to this function and its type will be detected, via the `_Generic` feature of ANSI C11. For a user-defined type, `x` is a `void *` pointer that points to a memory space holding a single entry of a scalar that has exactly the same user-defined type as the matrix `C`. This user-defined type must exactly match the user-defined type of `C` since no typecasting is done between user-defined types.

If a `void *` pointer is passed in and the type of the underlying scalar does not exactly match the user-defined type of `C`, then results are undefined. No error status will be returned since GraphBLAS has no way of catching this error. If `x` is a `GrB_Scalar` with no entry, then it is implicitly expanded into a matrix `A` of size `ni` by `nj`, with no entries present.

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GxB_BY_ROW`; if `|J|` is small, then it is fastest if the format of `C` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

10.9 GrB_assign: submatrix assignment

The methods described in this section are all variations of the form `C(I,J)=A`, which modifies a submatrix of the matrix `C`. All methods can be used in their generic form with the single name `GrB_assign`. These methods are very similar to their `GxB_subassign` counterparts in Section 10.8. They differ primarily in the size of the `Mask`, and how the `GrB_REPLACE` option works. Section 10.11 compares `GxB_subassign` and `GrB_assign`.

See Section 9 for a description of `I`, `ni`, `J`, and `nj`.

10.9.1 GrB_Vector_assign: assign to a subvector

```
GrB_Info GrB_assign                 // w<mask>(I) = accum (w(I),u)
(
    GrB_Vector w,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w(I),t)
    const GrB_Vector u,             // first input:  vector u
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_assign` operates on a subvector `w(I)` of `w`, modifying it with the vector `u`. The `mask` vector has the same size as `w`. The method is identical to `GrB_Matrix_assign` described in Section 10.9.2, where all matrices have a single column each. The only other difference is that the input `u` in this method is not transposed via the `GrB_INP0` descriptor.

10.9.2 GrB_Matrix_assign: assign to a submatrix

```
GrB_Info GrB_assign                 // C<Mask>(I,J) = accum (C(I,J),A)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C(I,J),T)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C, Mask, and A
) ;
```

`GrB_Matrix_assign` operates on a submatrix `S` of `C`, modifying it with the matrix `A`. It may also modify all of `C`, depending on the input descriptor `desc` and the `Mask`.

Step	GraphBLAS notation	description
1	$\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$	extract $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix
2	$\mathbf{S} = \mathbf{S} \odot \mathbf{A}$	apply the accumulator (but not the mask) to $\mathbf{S}$
3	$\mathbf{Z} = \mathbf{C}$	make a copy of $\mathbf{C}$
4	$\mathbf{Z}(\mathbf{I}, \mathbf{J}) = \mathbf{S}$	put the submatrix into $\mathbf{Z}(\mathbf{I}, \mathbf{J})$
5	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$	apply the mask/replace phase to all of $\mathbf{C}$

In contrast to `GxB_subassign`, the `Mask` has the same as `C`.

Step 1 extracts the submatrix and then Step 2 applies the accumulator (or $\mathbf{S} = \mathbf{A}$ if `accum` is `NULL`). The `Mask` is not yet applied.

Step 3 makes a copy of the $\mathbf{C}$ matrix, and then Step 4 writes the submatrix $\mathbf{S}$ into $\mathbf{Z}$. This is the same as Step 3 of `GxB_subassign`, except that it operates on a temporary matrix $\mathbf{Z}$.

Finally, Step 5 writes $\mathbf{Z}$ back into $\mathbf{C}$ via the `Mask`, using the Mask/Replace Phase described in Section 2.3. If `GrB_REPLACE` is enabled, then all of $\mathbf{C}$ is cleared prior to writing $\mathbf{Z}$ via the mask. As a result, the `GrB_REPLACE` option can delete entries outside the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix.

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GxB_BY_ROW`; if `|J|` is small, then it is fastest if the format of `C` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

10.9.3 GrB_Col_assign: assign to a sub-column of a matrix

```
GrB_Info GrB_assign                 // C<mask>(I,j) = accum (C(I,j),u)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(:,j), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(I,j),t)
    const GrB_Vector u,             // input vector
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index j,              // column index
    const GrB_Descriptor desc       // descriptor for C(:,j) and mask
) ;
```

`GrB_Col_assign` modifies a single sub-column of a matrix `C`. It is the same as `GrB_Matrix_assign` where the index vector `J[0]=j` is a single column index, and where all matrices in `GrB_Matrix_assign` (except `C`) consist of a single column.

Unlike `GrB_Matrix_assign`, the `mask` is a vector with the same size as a single column of `C`.

The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GrB_Matrix_assign` for further details.

Performance considerations: `GrB_Col_assign` is much faster than `GrB_Row_assign` if the format of `C` is `GxB_BY_COL`. `GrB_Row_assign` is much faster than `GrB_Col_assign` if the format of `C` is `GxB_BY_ROW`.

10.9.4 GrB_Row_assign: assign to a sub-row of a matrix

```
GrB_Info GrB_assign                 // C<mask'>(i,J) = accum (C(i,J),u')
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(i,:), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(i,J),t)
    const GrB_Vector u,             // input vector
    const GrB_Index i,              // row index
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C(i,:) and mask
) ;
```

`GrB_Row_assign` modifies a single sub-row of a matrix `C`. It is the same as `GrB_Matrix_assign` where the index vector `I[0]=i` is a single row index, and where all matrices in `GrB_Matrix_assign` (except `C`) consist of a single row.

Unlike `GrB_Matrix_assign`, the `mask` is a vector with the same size as a single row of `C`.

The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GrB_Matrix_assign` for further details.

Performance considerations: `GrB_Col_assign` is much faster than `GrB_Row_assign` if the format of `C` is `GxB_BY_COL`. `GrB_Row_assign` is much faster than `GrB_Col_assign` if the format of `C` is `GxB_BY_ROW`.

10.9.5 GrB_Vector_assign_<type>: assign a scalar to a subvector

```
GrB_Info GrB_assign                 // w<mask>(I) = accum (w(I),x)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w(I),x)
    const <type> x,                 // scalar to assign to w(I)
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_assign_<type>` assigns a single scalar to an entire subvector of the vector `w`. The operation is exactly like setting a single entry in an `n`-by-1 matrix, `A(I,0) = x`, where the column index for a vector is implicitly `j=0`. The `mask` vector has the same size as `w`. For further details of this function, see `GrB_Matrix_assign_<type>` in the next section (10.9.6).

Following the C API Specification, results are well-defined if `I` contains duplicate indices. Duplicate indices are simply ignored. See Section 10.10 for more details.

10.9.6 GrB_Matrix_assign_<type>: assign a scalar to a submatrix

```
GrB_Info GrB_assign                 // C<Mask>(I,J) = accum (C(I,J),x)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C(I,J),x)
    const <type> x,                 // scalar to assign to C(I,J)
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C and Mask
) ;
```

`GrB_Matrix_assign_<type>` assigns a single scalar to an entire submatrix of `C`, like the *scalar expansion* `C(I,J)=x` in MATLAB. The scalar `x` is implicitly expanded into a matrix `A` of size `ni` by `nj`, and then the matrix `A` is assigned to `C(I,J)` using the same method as in `GrB_Matrix_assign`. Refer to that function in Section 10.9.2 for further details.

The `Mask` has the same size as `C`.

For the accumulation step, the scalar `x` is typecasted directly into the type of `C` when the `accum` operator is not applied to it, or into the `ytype` of the `accum` operator, if `accum` is not NULL, for entries that are already present in `C`.

The `<type> x` notation is otherwise the same as `GrB_Matrix_setElement` (see Section 6.9.11). Any value can be passed to this function and its type will be detected, via the `_Generic` feature of ANSI C11. For a user-defined type, `x` is a `void *` pointer that points to a memory space holding a single entry of a scalar that has exactly the same user-defined type as the matrix `C`. This user-defined type must exactly match the user-defined type of `C` since no typecasting is done between user-defined types.

If a `void *` pointer is passed in and the type of the underlying scalar does not exactly match the user-defined type of `C`, then results are undefined. No error status will be returned since GraphBLAS has no way of catching this error.

If `x` is a `GrB_Scalar` with no entry, then it is implicitly expanded into a matrix `A` of size `ni` by `nj`, with no entries present.

Following the C API Specification, results are well-defined if `I` or `J` contain duplicate indices. Duplicate indices are simply ignored. See Section 10.10 for more details.

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GxB_BY_ROW`; if `|J|` is small, then it is fastest if the format of `C` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

10.10 Duplicate indices in GrB_assign and GxB_subassign

According to the GraphBLAS C API Specification if the index vectors `I` or `J` contain duplicate indices, the results are undefined for `GrB_Matrix_assign`, `GrB_Matrix_assign`, `GrB_Col_assign`, and `GrB_Row_assign`. Only the scalar assignment operations (`GrB_Matrix_assign_TYPE` and `GrB_Matrix_assign_TYPE`) are well-defined when duplicates appear in `I` and `J`. In those two functions, duplicate indices are ignored.

As an extension to the specification, SuiteSparse:GraphBLAS provides a definition of how duplicate indices are handled in all cases. If `I` has duplicate indices, they are ignored and the last unique entry in the list is used. When no mask and no accumulator is present, the results are identical to how MATLAB handles duplicate indices in the built-in expression `C(I,J)=A`. Details of how this is done is shown below.

```
function C = subassign (C, I, J, A)
% submatrix assignment with pre-sort of I and J; and remove duplicates

% delete duplicates from I, keeping the last one seen
[I2 I2k] = sort (I) ;
Idupl = [(I2 (1:end-1) == I2 (2:end)), false] ;
I2  = I2  (~Idupl) ;
I2k = I2k (~Idupl) ;
assert (isequal (I2, unique (I)))

% delete duplicates from J, keeping the last one seen
[J2 J2k] = sort (J) ;
Jdupl = [(J2 (1:end-1) == J2 (2:end)), false] ;
J2  = J2  (~Jdupl) ;
J2k = J2k (~Jdupl) ;
assert (isequal (J2, unique (J)))

% do the submatrix assignment, with no duplicates in I2 or J2
C (I2,J2) = A (I2k,J2k) ;
```

If a mask is present, then it is replaced with `M = M (I2k, J2k)` for `GxB_subassign`, or with `M = M (I2, J2)` for `GrB_assign`. If an accumulator operator is present, it is applied after the duplicates are removed, as (for example):

```
C (I2,J2) = C (I2,J2) + A (I2k,J2k) ;
```

These definitions allow the Octave/MATLAB interface to GraphBLAS to return the same results for `C(I,J)=A` for a `GrB` object as they do for built-in Octave/MATLAB matrices. They also allow the assignment to be done in parallel.

Results are always well-defined in SuiteSparse:GraphBLAS, but they might not be what you expect. For example, suppose the `MIN` operator is being used the following

assigment to the vector `x`, and suppose `I` contains the entries `[0 0]`. Suppose `x` is initially empty, of length 1, and suppose `y` is a vector of length 2 with the values `[5 7]`.

```
#include "GraphBLAS.h"
#include <stdio.h>
int main (void)
{
    GrB_init (GrB_NONBLOCKING) ;
    GrB_Vector x, y ;
    GrB_Vector_new (&x, GrB_INT32, 1) ;
    GrB_Vector_new (&y, GrB_INT32, 2) ;
    GrB_Index I [2] = {0, 0} ;
    GrB_Vector_setElement (y, 5, 0) ;
    GrB_Vector_setElement (y, 7, 1) ;
    GrB_Vector_wait (&y) ;
    GxB_print (x, 3) ;
    GxB_print (y, 3) ;
    GrB_assign (x, NULL, GrB_MIN_INT32, y, I, 2, NULL) ;
    GrB_Vector_wait (&y) ;
    GxB_print (x, 3) ;
    GrB_finalize ( ) ;
}
```

You might (wrongly) expect the result to be the vector `x(0)=5`, since two entries seem to be assigned, and the min operator might be expected to take the minimum of the two. This is not how SuiteSparse:GraphBLAS handles duplicates.

Instead, the first duplicate index of `I` is discarded (`I [0] = 0`, and `y(0)=5`). and only the second entry is used (`I [1] = 0`, and `y(1)=7`). The output of the above program is:

```
1x1 GraphBLAS int32_t vector, sparse by col:
x, no entries


2x1 GraphBLAS int32_t vector, sparse by col:
y, 2 entries

  (0,0)   5
  (1,0)   7
```

```
1x1 GraphBLAS int32_t vector, sparse by col:
x, 1 entry

  (0,0)   7
```

You see that the result is `x(0)=7`, since the `y(0)=5` entry has been ignored because of the duplicate indices in `I`.

SPEC: Providing a well-defined behavior for duplicate indices with matrix and vector assignment is an extension to the specification. The specification only defines the behavior when assigning a scalar into a matrix or vector, and states that duplicate indices otherwise lead to undefined results.

10.11 Comparing GrB_assign and GxB_subassign

The GxB_subassign and GrB_assign operations are very similar, but they differ in two ways:

1. **The Mask has a different size:** The mask in GxB_subassign has the same dimensions as w(I) for vectors and C(I,J) for matrices. In GrB_assign, the mask is the same size as w or C, respectively (except for the row/col variants). The two masks are related. If M is the mask for GrB_assign, then M(I,J) is the mask for GxB_subassign. If there is no mask, or if I and J are both GrB_ALL, the two masks are the same. For GrB_Row_assign and GrB_Col_assign, the mask vector is the same size as a row or column of C, respectively. For the corresponding GxB_Row_subassign and GxB_Col_subassign operations, the mask is the same size as the sub-row C(i,J) or subcolumn C(I,j), respectively.

2. GrB_REPLACE **is different:** They differ in how C is affected in areas outside the C(I,J) submatrix. In GxB_subassign, the C(I,J) submatrix is the only part of C that can be modified, and no part of C outside the submatrix is ever modified. In GrB_assign, it is possible to delete entries in C outside the submatrix, but only in one specific manner. Suppose the mask M is present (or, suppose it is not present but GrB_COMP is true). After (optionally) complementing the mask, the value of M(i,j) can be 0 for some entry outside the C(I,J) submatrix. If the GrB_REPLACE descriptor is true, GrB_assign deletes this entry.

GxB_subassign and GrB_assign are identical if GrB_REPLACE is set to its default value of false, and if the masks happen to be the same. The two masks can be the same in two cases: either the Mask input is NULL (and it is not complemented via GrB_COMP), or I and J are both GrB_ALL. If all these conditions hold, the two algorithms are identical and have the same performance. Otherwise, GxB_subassign is much faster than GrB_assign when the latter must examine the entire matrix C to delete entries (when GrB_REPLACE is true), and if it must deal with a much larger Mask matrix. However, both methods have specific uses.

Consider using C(I,J)+=F for many submatrices F (for example, when assembling a finite-element matrix). If the Mask is meant as a specification for which entries of C should appear in the final result, then use GrB_assign.

If instead the Mask is meant to control which entries of the submatrix C(I,J) are modified by the finite-element F, then use GxB_subassign. This is particularly useful is the Mask is a template that follows along with the finite-element F, independent of where it is applied to C. Using GrB_assign would be very difficult in this case since a new Mask, the same size as C, would need to be constructed for each finite-element F.

In GraphBLAS notation, the two methods can be described as follows:

matrix and vector subassign	$\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{I},\mathbf{J}) \odot \mathbf{A}$
matrix and vector assign	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I},\mathbf{J}) = \mathbf{C}(\mathbf{I},\mathbf{J}) \odot \mathbf{A}$

This notation does not include the details of the GrB_COMP and GrB_REPLACE descriptors, but it does illustrate the difference in the Mask. In the subassign, Mask is the same size as C(I,J) and A. If I[0]=i and J[0]=j, Then Mask(0,0) controls how C(i,j) is

modified by the subassign, from the value `A(0,0)`. In the assign, `Mask` is the same size as `C`, and `Mask(i,j)` controls how `C(i,j)` is modified.

The `GxB_subassign` and `GrB_assign` functions have the same signatures; they differ only in how they consider the `Mask` and the `GrB_REPLACE` descriptor

Details of each step of the two operations are listed below:

Step	`GrB_Matrix_assign`	`GxB_Matrix_subassign`
1	$\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$	$\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$
2	$\mathbf{S} = \mathbf{S} \odot \mathbf{A}$	$\mathbf{S}\langle\mathbf{M}\rangle = \mathbf{S} \odot \mathbf{A}$
3	$\mathbf{Z} = \mathbf{C}$	$\mathbf{C}(\mathbf{I}, \mathbf{J}) = \mathbf{S}$
4	$\mathbf{Z}(\mathbf{I}, \mathbf{J}) = \mathbf{S}$	
5	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$	

Step 1 is the same. In the Accumulator Phase (Step 2), the expression $\mathbf{S} \odot \mathbf{A}$, described in Section 2.3, is the same in both operations. The result is simply $\mathbf{A}$ if `accum` is `NULL`. It only applies to the submatrix $\mathbf{S}$, not the whole matrix. The result $\mathbf{S} \odot \mathbf{A}$ is used differently in the Mask/Replace phase.

The Mask/Replace Phase, described in Section 2.3 is different:

- For `GrB_assign` (Step 5), the mask is applied to all of $\mathbf{C}$. The mask has the same size as $\mathbf{C}$. Just prior to making the assignment via the mask, the `GrB_REPLACE` option can be used to clear all of $\mathbf{C}$ first. This is the only way in which entries in $\mathbf{C}$ that are outside the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix can be modified by this operation.
- For `GxB_subassign` (Step 2b), the mask is applied to just $\mathbf{S}$. The mask has the same size as $\mathbf{C}(\mathbf{I}, \mathbf{J})$, $\mathbf{S}$, and $\mathbf{A}$. Just prior to making the assignment via the mask, the `GrB_REPLACE` option can be used to clear $\mathbf{S}$ first. No entries in $\mathbf{C}$ that are outside the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ can be modified by this operation. Thus, `GrB_REPLACE` has no effect on entries in $\mathbf{C}$ outside the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix.

The differences between `GrB_assign` and `GxB_subassign` can be seen in Tables 5 and 6. The first table considers the case when the entry c_{ij} is in the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix, and it describes what is computed for both `GrB_assign` and `GxB_subassign`. They perform the exact same computation; the only difference is how the value of the mask is specified. Compare Table 5 with Table 1 in Section 7.

The first column of Table 5 is *yes* if `GrB_REPLACE` is enabled, and a dash otherwise. The second column is *yes* if an accumulator operator is given, and a dash otherwise. The third column is c_{ij} if the entry is present in $\mathbf{C}$, and a dash otherwise. The fourth column is $a_{i'j'}$ if the corresponding entry is present in $\mathbf{A}$, where $i = \mathbf{I}(i')$ and $j = \mathbf{J}(i')$.

The *mask* column is 1 if the effective value of the mask mask allows $\mathbf{C}$ to be modified, and 0 otherwise. This is m_{ij} for `GrB_assign`, and $m_{i'j'}$ for `GxB_subassign`, to reflect the difference in the mask, but this difference is not reflected in the table. The value 1 or 0 is the value of the entry in the mask after it is optionally complemented via the `GrB_COMP` option.

Finally, the last column is the action taken in this case. It is left blank if no action is taken, in which case c_{ij} is not modified if present, or not inserted into $\mathbf{C}$ if not present.

repl	accum	**C**	**A**	mask	action taken by `GrB_assign` and `GxB_subassign`
-	-	c_{ij}	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, update
-	-	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
-	-	c_{ij}	-	1	delete c_{ij} because $a_{i'j'}$ not present
-	-	-	-	1	
-	-	c_{ij}	$a_{i'j'}$	0	
-	-	-	$a_{i'j'}$	0	
-	-	c_{ij}	-	0	
-	-	-	-	0	
yes	-	c_{ij}	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, update
yes	-	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
yes	-	c_{ij}	-	1	delete c_{ij} because $a_{i'j'}$ not present
yes	-	-	-	1	
yes	-	c_{ij}	$a_{i'j'}$	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	$a_{i'j'}$	0	
yes	-	c_{ij}	-	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	-	0	
-	yes	c_{ij}	$a_{i'j'}$	1	$c_{ij} = c_{ij} \odot a_{i'j'}$, apply accumulator
-	yes	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
-	yes	c_{ij}	-	1	
-	yes	-	-	1	
-	yes	c_{ij}	$a_{i'j'}$	0	
-	yes	-	$a_{i'j'}$	0	
-	yes	c_{ij}	-	0	
-	yes	-	-	0	
yes	yes	c_{ij}	$a_{i'j'}$	1	$c_{ij} = c_{ij} \odot a_{i'j'}$, apply accumulator
yes	yes	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
yes	yes	c_{ij}	-	1	
yes	yes	-	-	1	
yes	yes	c_{ij}	$a_{i'j'}$	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	$a_{i'j'}$	0	
yes	yes	c_{ij}	-	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	-	0	

Table 5: Results of assign and subassign for entries in the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix

repl	accum	$\mathbf{C}$	$\mathbf{C} = \mathbf{Z}$	mask	action taken by `GrB_assign`
-	-	c_{ij}	c_{ij}	1	
-	-	-	-	1	
-	-	c_{ij}	c_{ij}	0	
-	-	-	-	0	
yes	-	c_{ij}	c_{ij}	1	
yes	-	-	-	1	
yes	-	c_{ij}	c_{ij}	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	-	0	
-	yes	c_{ij}	c_{ij}	1	
-	yes	-	-	1	
-	yes	c_{ij}	c_{ij}	0	
-	yes	-	-	0	
yes	yes	c_{ij}	c_{ij}	1	
yes	yes	-	-	1	
yes	yes	c_{ij}	c_{ij}	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	-	0	

Table 6: Results of assign for entries outside the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix. Subassign has no effect on these entries.

Table 6 illustrates how `GrB_assign` and `GxB_subassign` differ for entries outside the submatrix. `GxB_subassign` never modifies any entry outside the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix, but `GrB_assign` can modify them in two cases listed in Table 6. When the `GrB_REPLACE` option is selected, and when the `Mask(i,j)` for an entry c_{ij} is false (or if the `Mask(i,j)` is true and `GrB_COMP` is enabled via the descriptor), then the entry is deleted by `GrB_assign`.

The fourth column of Table 6 differs from Table 5, since entries in $\mathbf{A}$ never affect these entries. Instead, for all index pairs outside the $I \times J$ submatrix, $\mathbf{C}$ and $\mathbf{Z}$ are identical (see Step 3 above). As a result, each section of the table includes just two cases: either c_{ij} is present, or not. This in contrast to Table 5, where each section must consider four different cases.

The `GrB_Row_assign` and `GrB_Col_assign` operations are slightly different. They only affect a single row or column of $\mathbf{C}$. For `GrB_Row_assign`, Table 6 only applies to entries in the single row `C(i,J)` that are outside the list of indices, `J`. For `GrB_Col_assign`, Table 6 only applies to entries in the single column `C(I,j)` that are outside the list of indices, `I`.

10.11.1 Example

The difference between `GxB_subassign` and `GrB_assign` is illustrated in the following example. Consider the 2-by-2 matrix $\mathbf{C}$ where all entries are present.

$$\mathbf{C} = \begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix}$$

Suppose `GrB_REPLACE` is true, and `GrB_COMP` is false. Let the `Mask` be:

$$\mathbf{M} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Let $\mathbf{A} = 100$, and let the index sets be $\mathbf{I} = 0$ and $\mathbf{J} = 1$. Consider the computation $\mathbf{C}\langle\mathbf{M}\rangle(0,1) = \mathbf{C}(0,1) + \mathbf{A}$, using the `GrB_assign` operation. The result is:

$$\mathbf{C} = \begin{bmatrix} 11 & 112 \\ - & 22 \end{bmatrix}.$$

The $(0,1)$ entry is updated and the $(1,0)$ entry is deleted because its `Mask` is zero. The other two entries are not modified since $\mathbf{Z} = \mathbf{C}$ outside the submatrix, and those two values are written back into $\mathbf{C}$ because their `Mask` values are 1. The $(1,0)$ entry is deleted because the entry $\mathbf{Z}(1,0) = 21$ is prevented from being written back into $\mathbf{C}$ since `Mask(1,0)=0`.

Now consider the analogous `GxB_subassign` operation. The `Mask` has the same size as $\mathbf{A}$, namely:

$$\mathbf{M} = \begin{bmatrix} 1 \end{bmatrix}.$$

After computing $\mathbf{C}(0,1)\langle\mathbf{M}\rangle = \mathbf{C}(0,1) + \mathbf{A}$, the result is

$$\mathbf{C} = \begin{bmatrix} 11 & 112 \\ 21 & 22 \end{bmatrix}.$$

Only the $\mathbf{C}(\mathbf{I},\mathbf{J})$ submatrix, the single entry $\mathbf{C}(0,1)$, is modified by `GxB_subassign`. The entry $\mathbf{C}(1,0) = 21$ is unaffected by `GxB_subassign`, but it is deleted by `GrB_assign`.

10.11.2 Performance of GxB_subassign, GrB_assign and GrB_*_setElement

When SuiteSparse:GraphBLAS uses non-blocking mode, the modifications to a matrix by `GxB_subassign`, `GrB_assign`, and `GrB_*_setElement` can postponed, and computed all at once later on. This has a huge impact on performance.

A sequence of assignments is fast if their completion can be postponed for as long as possible, or if they do not modify the pattern at all. Modifying the pattern can be costly, but it is fast if non-blocking mode can be fully exploited.

Consider a sequence of t submatrix assignments `C(I,J)=C(I,J)+A` to an n-by-n matrix `C` where each submatrix `A` has size a-by-a with s entries, and where `C` starts with c entries. Assume the matrices are all stored in non-hypersparse form, by row (`GxB_BY_ROW`).

If blocking mode is enabled, or if the sequence requires the matrix to be completed after each assignment, each of the t assignments takes $O(a + s \log n)$ time to process the `A` matrix and then $O(n+c+s \log s)$ time to complete `C`. The latter step uses `GrB_*_build` to build an update matrix and then merge it with `C`. This step does not occur if the sequence of assignments does not add new entries to the pattern of `C`, however. Assuming in the worst case that the pattern does change, the total time is $O(t\,[a + s \log n + n + c + s \log s])$.

If the sequence can be computed with all updates postponed until the end of the sequence, then the total time is no worse than $O(a + s \log n)$ to process each `A` matrix, for t assignments, and then a single `build` at the end, taking $O(n + c + st \log st)$ time. The total time is $O(t\,[a + s \log n] + (n + c + st \log st))$. If no new entries appear in `C` the time drops to $O(t\,[a + s \log n])$, and in this case, the time for both methods is the same; both are equally efficient.

A few simplifying assumptions are useful to compare these times. Consider a graph of n nodes with $O(n)$ edges, and with a constant bound on the degree of each node. The asymptotic bounds assume a worst-case scenario where `C` has a least some dense rows (thus the $\log n$ terms). If these are not present, if both t and c are $O(n)$, and if a and s are constants, then the total time with blocking mode becomes $O(n^2)$, assuming the pattern of `C` changes at each assignment. This very high for a sparse graph problem. In contrast, the non-blocking time becomes $O(n \log n)$ under these same assumptions, which is asymptotically much faster.

The difference in practice can be very dramatic, since n can be many millions for sparse graphs with n nodes and $O(n)$, which can be handled on a commodity laptop.

The following guidelines should be considered when using `GxB_subassign`, `GrB_assign` and `GrB_*_setElement`.

1. A sequence of assignments that does not modify the pattern at all is fast, taking as little as $\Omega(1)$ time per entry modified. The worst case time complexity is $O(\log n)$ per entry, assuming they all modify a dense row of `C` with `n` entries, which can occur in practice. It is more common, however, that most rows of `C` have a constant number of entries, independent of `n`. No work is ever left pending when the pattern of `C` does not change.

2. A sequence of assignments that modifies the entries that already exist in the pattern of a matrix, or adds new entries to the pattern (using the same `accum` operator), but does not delete any entries, is fast. The matrix is not completed until the end of the sequence.

3. Similarly, a sequence that modifies existing entries, or deletes them, but does not add new ones, is also fast. This sequence can also repeatedly delete pre-existing entries and then reinstate them and still be fast. The matrix is not completed until the end of the sequence.

4. A sequence that mixes assignments of types (2) and (3) above can be costly, since the matrix may need to be completed after each assignment. The time complexity can become quadratic in the worst case.

5. However, any single assignment takes no more than $O(a + s \log n + n + c + s \log s)$ time, even including the time for a matrix completion, where `C` is n-by-n with c entries and `A` is a-by-a with s entries. This time is essentially linear in the size of the matrix `C`, if `A` is relatively small and sparse compared with `C`. In this case, $n + c$ are the two dominant terms.

6. In general, `GxB_subassign` is faster than `GrB_assign`. If `GrB_REPLACE` is used with `GrB_assign`, the entire matrix `C` must be traversed. This is much slower than `GxB_subassign`, which only needs to examine the `C(I,J)` submatrix. Furthermore, `GrB_assign` must deal with a much larger `Mask` matrix, whereas `GxB_subassign` has a smaller mask. Since its mask is smaller, `GxB_subassign` takes less time than `GrB_assign` to access the mask.

Submatrix assignment in SuiteSparse:GraphBLAS is extremely efficient, even without considering the advantages of non-blocking mode discussed in Section 10.11. It can be up to 1000x faster than MATLAB R2019b, or even higher depending on the kind of matrix assignment. MATLAB logical indexing (the mask of GraphBLAS) is extremely faster with GraphBLAS as compared in MATLAB R2019b; differences of up to 250,000x have been observed (0.4 seconds in GraphBLAS versus 28 hours in MATLAB).

All of the 28 variants (each with their own source code) are either asymptotically optimal, or to within a log factor of being asymptotically optimal. The methods are also fully parallel. For hypersparse matrices, the term n in the expressions in the above discussion is dropped, and is replaced with $h \log h$, at the worst case, where $h << n$ is the

number of non-empty columns of a hypersparse matrix stored by column, or the number of non-empty rows of a hypersparse matrix stored by row. In many methods, n is replaced with h, not $h \log h$.

10.12 GrB_apply: apply a unary, binary, or index-unary operator

`GrB_apply` is the generic name for 92 specific functions:

- `GrB_Vector_apply` and `GrB_Matrix_apply` apply a unary operator to the entries of a matrix (two variants).
- `GrB_*_apply_BinaryOp1st_*` applies a binary operator where a single scalar is provided as the x input to the binary operator. There are 30 variants, depending on the type of the scalar: (matrix or vector) x (13 built-in types, one for user-defined types, and a version for `GrB_Scalar`).
- `GrB_*_apply_BinaryOp2nd_*` applies a binary operator where a single scalar is provided as the y input to the binary operator. There are 30 variants, depending on the type of the scalar: (matrix or vector) x (13 built-in types, one for user-defined types, and a version for `GrB_Scalar`).
- `GrB_*_apply_IndexOp_*` applies a `GrB_IndexUnaryOp`, single scalar is provided as the scalar y input to the index-unary operator. There are 30 variants, depending on the type of the scalar: (matrix or vector) x (13 built-in types, one for user-defined types, and a version for `GrB_Scalar`).

The generic name appears in the function prototypes, but the specific function name is used when describing each variation. When discussing features that apply to all versions, the simple name `GrB_apply` is used.

10.12.1 GrB_Vector_apply: apply a unary operator to a vector

```
GrB_Info GrB_apply                  // w<mask> = accum (w, op(u))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_UnaryOp op,           // operator to apply to the entries
    const GrB_Vector u,             // first input:  vector u
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_apply` applies a unary operator to the entries of a vector, analogous to `t = op(u)` in MATLAB except the operator `op` is only applied to entries in the pattern of `u`. Implicit values outside the pattern of `u` are not affected. The entries in `u` are typecasted into the `xtype` of the unary operator. The vector `t` has the same type as the `ztype` of the unary operator. The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

10.12.2 GrB_Matrix_apply: apply a unary operator to a matrix

```
GrB_Info GrB_apply                  // C<Mask> = accum (C, op(A)) or op(A')
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_UnaryOp op,           // operator to apply to the entries
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

`GrB_Matrix_apply` applies a unary operator to the entries of a matrix, analogous to `T = op(A)` in MATLAB except the operator `op` is only applied to entries in the pattern of `A`. Implicit values outside the pattern of `A` are not affected. The input matrix `A` may be transposed first. The entries in `A` are typecasted into the `xtype` of the unary operator. The matrix `T` has the same type as the `ztype` of the unary operator. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

The built-in `GrB_IDENTITY_`T operators (one for each built-in type T) are very useful when combined with this function, enabling it to compute $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}$. This makes `GrB_apply` a direct interface to the accumulator/mask function for both matrices and vectors. The `GrB_IDENTITY_`T operators also provide the fastest stand-alone typecasting methods in SuiteSparse:GraphBLAS, with all $13 \times 13 = 169$ methods appearing as individual functions, to typecast between any of the 13 built-in types.

To compute $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}$ or $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}$ for user-defined types, the user application would need to define an identity operator for the type. Since GraphBLAS cannot detect that it is an identity operator, it must call the operator to make the full copy `T=A` and apply the operator to each entry of the matrix or vector.

The other GraphBLAS operation that provides a direct interface to the accumulator/mask function is `GrB_transpose`, which does not require an operator to perform this task. As a result, `GrB_transpose` can be used as an efficient and direct interface to the accumulator/mask function for both built-in and user-defined types. However, it is only available for matrices, not vectors.

10.12.3 GrB_Vector_apply_BinaryOp1st: apply a binary operator to a vector; 1st scalar binding

```
GrB_Info GrB_apply                  // w<mask> = accum (w, op(x,u))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_BinaryOp op,          // operator to apply to the entries
    <type> x,                       // first input:  scalar x
    const GrB_Vector u,             // second input: vector u
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_apply_BinaryOp1st_<type>` applies a binary operator $z = f(x, y)$ to a vector, where a scalar x is bound to the first input of the operator. The scalar `x` can be a non-opaque C scalar corresponding to a built-in type, a `void *` for user-defined types, or a `GrB_Scalar`. It is otherwise identical to `GrB_Vector_apply`.

10.12.4 GrB_Vector_apply_BinaryOp2nd: apply a binary operator to a vector; 2nd scalar binding

```
GrB_Info GrB_apply                  // w<mask> = accum (w, op(u,y))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_BinaryOp op,          // operator to apply to the entries
    const GrB_Vector u,             // first input:  vector u
    <type> y,                       // second input: scalar y
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_apply_BinaryOp2nd_<type>` applies a binary operator $z = f(x, y)$ to a vector, where a scalar y is bound to the second input of the operator. The scalar `x` can be a non-opaque C scalar corresponding to a built-in type, a `void *` for user-defined types, or a `GrB_Scalar`. It is otherwise identical to `GrB_Vector_apply`.

10.12.5 GrB_Vector_apply_IndexOp: apply an index-unary operator to a vector

```
GrB_Info GrB_apply                  // w<mask> = accum (w, op(u,y))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_IndexUnaryOp op,      // operator to apply to the entries
    const GrB_Vector u,             // first input:  vector u
    const <type> y,                 // second input: scalar y
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_apply_IndexOp_<type>` applies an index-unary operator $z = f(x, i, 0, y)$ to a vector. The scalar `y` can be a non-opaque C scalar corresponding to a built-in type, a `void *` for user-defined types, or a `GrB_Scalar`. It is otherwise identical to `GrB_Vector_apply`.

10.12.6 GrB_Matrix_apply_BinaryOp1st: apply a binary operator to a matrix; 1st scalar binding

```
GrB_Info GrB_apply                  // C<M>=accum(C,op(x,A))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_BinaryOp op,          // operator to apply to the entries
    <type> x,                       // first input:  scalar x
    const GrB_Matrix A,             // second input: matrix A
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

`GrB_Matrix_apply_BinaryOp1st_<type>` applies a binary operator $z = f(x, y)$ to a matrix, where a scalar x is bound to the first input of the operator. The scalar `x` can be a non-opaque C scalar corresponding to a built-in type, a `void *` for user-defined types, or a `GrB_Scalar`. It is otherwise identical to `GrB_Matrix_apply`.

10.12.7 GrB_Matrix_apply_BinaryOp2nd: apply a binary operator to a matrix; 2nd scalar binding

```
GrB_Info GrB_apply                  // C<M>=accum(C,op(A,y))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_BinaryOp op,          // operator to apply to the entries
    const GrB_Matrix A,             // first input:  matrix A
    <type> y,                       // second input: scalar y
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

`GrB_Matrix_apply_BinaryOp2nd_<type>` applies a binary operator $z = f(x, y)$ to a matrix, where a scalar x is bound to the second input of the operator. The scalar `y` can be a non-opaque C scalar corresponding to a built-in type, a `void *` for user-defined types, or a `GrB_Scalar`. It is otherwise identical to `GrB_Matrix_apply`.

10.12.8 GrB_Matrix_apply_IndexOp: apply an index-unary operator to a matrix

```
GrB_Info GrB_apply                  // C<M>=accum(C,op(A,y))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_IndexUnaryOp op,      // operator to apply to the entries
    const GrB_Matrix A,             // first input:  matrix A
    const <type> y,                 // second input: scalar y
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

`GrB_Matrix_apply_IndexOp_<type>` applies an index-unary operator $z = f(x, i, j, y)$ to a matrix. The scalar `y` can be a non-opaque C scalar corresponding to a built-in type, a `void *` for user-defined types, or a `GrB_Scalar`. It is otherwise identical to `GrB_Matrix_apply`.

10.13 GrB_select: select entries based on an index-unary operator

The `GrB_select` function is the generic name for 30 specific functions, depending on whether it operates on a matrix or vector, and depending on the type of the scalar `y`: (matrix or vector) x (13 built-in types, `void *` for user-defined types, and a `GrB_Scalar`). The generic name appears in the function prototypes, but the specific function name is used when describing each variation. When discussing features that apply to both versions, the simple name `GrB_select` is used.

10.13.1 GrB_Vector_select: select entries from a vector

```
GrB_Info GrB_select                 // w<mask> = accum (w, op(u))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_IndexUnaryOp op,      // operator to apply to the entries
    const GrB_Vector u,             // first input:  vector u
    const <type> y,                 // second input: scalar y
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_select_*` applies a `GrB_IndexUnaryOp` operator to the entries of a vector. If the operator evaluates as `true` for the entry `u(i)`, it is copied to the vector `t`, or not copied if the operator evaluates to `false`. The vector `t` is then written to the result `w` via the mask/accumulator step. This operation operates on vectors just as if they were `m`-by-1 matrices, except that GraphBLAS never transposes a vector via the descriptor. Refer to the next section (10.13.2) on `GrB_Matrix_select` for more details.

10.13.2 GrB_Matrix_select: apply a select operator to a matrix

```
GrB_Info GrB_select                 // C<M>=accum(C,op(A))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_IndexUnaryOp op,      // operator to apply to the entries
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Scalar y,             // second input: scalar y
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

`GrB_Matrix_select_*` applies a `GrB_IndexUnaryOp` operator to the entries of a vector. If the operator evaluates as `true` for the entry `A(i,j)`, it is copied to the matrix `T`, or not copied if the operator evaluates to `false`. The input matrix `A` may be transposed first. The entries in `A` are typecasted into the `xtype` of the select operator. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

The matrix `T` has the same size and type as `A` (or the transpose of `A` if the input is transposed via the descriptor). The entries of `T` are a subset of those of `A`. Each entry `A(i,j)` of `A` is passed to the `op`, as $z = f(a_{ij}, i, j, y)$. If `A` is transposed first then the operator is applied to entries in the transposed matrix, `A'`. If z is returned as true, then the entry is copied into `T`, unchanged. If it returns false, the entry does not appear in `T`.

The action of `GrB_select` with the built-in index-unary operators is described in the table below. The MATLAB analogs are precise for `tril` and `triu`, but shorthand for the other operations. The MATLAB `diag` function returns a column with the diagonal, if `A` is a matrix, whereas the matrix `T` in `GrB_select` always has the same size as `A` (or its transpose if the `GrB_INP0` is set to `GrB_TRAN`). In the MATLAB analog column, `diag` is as if it operates like `GrB_select`, where `T` is a matrix.

The following operators may be used on matrices with a user-defined type: `GrB_ROWINDEX_*`, `GrB_COLINDEX_*`, `GrB_DIAGINDEX_*`, `GrB_TRIL`, `GrB_TRIU`, `GrB_DIAG`, `GrB_OFFIAG`, `GrB_COLLE`, `GrB_COLGT`, `GrB_ROWLE`, and `GrB_ROWGT`.

For floating-point values, comparisons with `NaN` always return false. The `GrB_VALUE*` operators should not be used with a scalar `y` that is equal to `NaN`. For this case, create a user-defined select operator that performs the test with the ANSI C `isnan` function instead.

GraphBLAS name	Octave/MATLAB analog	description
GrB_ROWINDEX_*	z=i+y	select A(i,j) if i != -y
GrB_COLINDEX_*	z=j+y	select A(i,j) if j != -y
GrB_DIAGINDEX_*	z=j-(i+y)	select A(i,j) if j != i+y
GrB_TRIL	z=(j<=(i+y))	select entries on or below the yth diagonal
GrB_TRIU	z=(j>=(i+y))	select entries on or above the yth diagonal
GrB_DIAG	z=(j==(i+y))	select entries on the yth diagonal
GrB_OFFDIAG	z=(j!=(i+y))	select entries not on the yth diagonal
GrB_COLLE	z=(j<=y)	select entries in columns 0 to y
GrB_COLGT	z=(j>y)	select entries in columns y+1 and above
GrB_ROWLE	z=(i<=y)	select entries in rows 0 to y
GrB_ROWGT	z=(i>y)	select entries in rows y+1 and above
GrB_VALUENE_T	z=(aij!=y)	select A(i,j) if it is not equal to y
GrB_VALUEEQ_T	z=(aij==y)	select A(i,j) is it equal to y
GrB_VALUEGT_T	z=(aij>y)	select A(i,j) is it greater than y
GrB_VALUEGE_T	z=(aij>=y)	select A(i,j) is it greater than or equal to y
GrB_VALUELT_T	z=(aij<y)	select A(i,j) is it less than y
GrB_VALUELE_T	z=(aij<=y)	select A(i,j) is it less than or equal to y

10.14 GrB_reduce: reduce to a vector or scalar

The generic function name `GrB_reduce` may be used for all specific functions discussed in this section. When the details of a specific function are discussed, the specific name is used for clarity.

SPEC: All methods below use a monoid for the reduction. The Specification also allows reductions using an associative and commutative binary operator. SuiteSparse:GraphBLAS permits the use of a `GrB_BinaryOp` instead of a `GrB_Monoid`, but only if the binary operator is built-in and corresponds to a known built-in monoid. For example, the binary operator `GrB_PLUS_FP64` can be used, since this is the binary operator of the built-in `GrB_PLUS_MONOID_FP64`. For other binary ops (including any user-defined ones), `GrB_NOT_IMPLEMENTED` is returned.

10.14.1 GrB_Matrix_reduce_Monoid reduce a matrix to a vector

```
GrB_Info GrB_reduce                 // w<mask> = accum (w,reduce(A))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_Monoid monoid,        // reduce monoid for t=reduce(A)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Descriptor desc       // descriptor for w, mask, and A
) ;
```

`GrB_Matrix_reduce_Monoid` reduces a matrix to a column vector using a monoid, roughly analogous to `t = sum (A')` in MATLAB, in the default case, where `t` is a column vector. By default, the method reduces across the rows to obtain a column vector; use `GrB_TRAN` to reduce down the columns.

The input matrix `A` may be transposed first. Its entries are then typecast into the type of the `reduce` operator or monoid. The reduction is applied to all entries in `A (i,:)` to produce the scalar `t (i)`. This is done without the use of the identity value of the monoid. If the `i`th row `A (i,:)` has no entries, then `(i)` is not an entry in `t` and its value is implicit. If `A (i,:)` has a single entry, then that is the result `t (i)` and `reduce` is not applied at all for the `i`th row. Otherwise, multiple entries in row `A (i,:)` are reduced via the `reduce` operator or monoid to obtain a single scalar, the result `t (i)`.

The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

10.14.2 GrB_Vector_reduce_<type>: reduce a vector to a scalar

```
GrB_Info GrB_reduce                 // c = accum (c, reduce_to_scalar (u))
(
    <type> *c,                      // result scalar
    const GrB_BinaryOp accum,       // optional accum for c=accum(c,t)
    const GrB_Monoid monoid,        // monoid to do the reduction
    const GrB_Vector u,             // vector to reduce
    const GrB_Descriptor desc       // descriptor (currently unused)
) ;

GrB_Info GrB_reduce                 // c = accum (c, reduce_to_scalar (u))
(
    GrB_Scalar c,                   // result scalar
    const GrB_BinaryOp accum,       // optional accum for c=accum(c,t)
    const GrB_Monoid monoid,        // monoid to do the reduction
    const GrB_Vector u,             // vector to reduce
    const GrB_Descriptor desc       // descriptor (currently unused)
) ;
```

`GrB_Vector_reduce_<type>` reduces a vector to a scalar, analogous to `t = sum (u)` in MATLAB, except that in GraphBLAS any commutative and associative monoid can be used in the reduction.

The scalar `c` can be a pointer C type: `bool`, `int8_t`, ... `float`, `double`, or `void *` for a user-defined type, or a `GrB_Scalar`. If `c` is a `void *` pointer to a user-defined type, the type must be identical to the type of the vector `u`. This cannot be checked by GraphBLAS and thus results are undefined if the types are not the same.

If the vector `u` has no entries, that identity value of the `monoid` is copied into the scalar `t` (unless `c` is a `GrB_Scalar`, in which case `t` is an empty `GrB_Scalar`, with no entry). Otherwise, all of the entries in the vector are reduced to a single scalar using the `monoid`.

The descriptor is unused, but it appears in case it is needed in future versions of the GraphBLAS API. This function has no mask so its accumulator/mask step differs from the other GraphBLAS operations. It does not use the methods described in Section 2.3, but uses the following method instead.

If `accum` is `NULL`, then the scalar `t` is typecast into the type of `c`, and `c = t` is the final result. Otherwise, the scalar `t` is typecast into the `ytype` of the `accum` operator, and the value of `c` (on input) is typecast into the `xtype` of the `accum` operator. Next, the scalar `z = accum (c,t)` is computed, of the `ztype` of the `accum` operator. Finally, `z` is typecast into the final result, `c`.

If `c` is a non-opaque scalar, no error message can be returned by `GrB_error`. If `c` is a `GrB_Scalar`, then `GrB_error(&err,c)` can be used to return an error string, if an error occurs.

10.14.3 GrB_Matrix_reduce_<type>: reduce a matrix to a scalar

```
GrB_Info GrB_reduce                 // c = accum (c, reduce_to_scalar (A))
(
    <type> *c,                      // result scalar
    const GrB_BinaryOp accum,       // optional accum for c=accum(c,t)
    const GrB_Monoid monoid,        // monoid to do the reduction
    const GrB_Matrix A,             // matrix to reduce
    const GrB_Descriptor desc       // descriptor (currently unused)
) ;

GrB_Info GrB_reduce                 // c = accum (c, reduce_to_scalar (A))
(
    GrB_Scalar c,                   // result scalar
    const GrB_BinaryOp accum,       // optional accum for c=accum(c,t)
    const GrB_Monoid monoid,        // monoid to do the reduction
    const GrB_Matrix A,             // matrix to reduce
    const GrB_Descriptor desc       // descriptor (currently unused)
) ;
```

`GrB_Matrix_reduce_<type>` reduces a matrix `A` to a scalar, roughly analogous to `t = sum (A (:))` in MATLAB. This function is identical to reducing a vector to a scalar, since the positions of the entries in a matrix or vector have no effect on the result. Refer to the reduction to scalar described in the previous Section 10.14.2.

10.15 GrB_transpose: transpose a matrix

```
GrB_Info GrB_transpose              // C<Mask> = accum (C, A')
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Descriptor desc       // descriptor for C, Mask, and A
) ;
```

`GrB_transpose` transposes a matrix `A`, just like the array transpose `T = A.'` in MATLAB. The internal result matrix `T = A'` (or merely `T = A` if `A` is transposed via the descriptor) has the same type as `A`. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3, which typecasts `T` as needed and applies the mask and accumulator.

To be consistent with the rest of the GraphBLAS API regarding the descriptor, the input matrix `A` may be transposed first by setting the `GrB_INP0` setting to `GrB_TRAN`. This results in a double transpose, and thus `A` is not transposed is computed.

10.16 GrB_kronecker: Kronecker product

```
GrB_Info GrB_kronecker              // C<Mask> = accum (C, kron(A,B))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const <operator> op,            // defines '*' for T=kron(A,B)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Matrix B,             // second input: matrix B
    const GrB_Descriptor desc       // descriptor for C, Mask, A, and B
) ;
```

`GrB_kronecker` computes the Kronecker product, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathrm{kron}(\mathbf{A}, \mathbf{B})$ where

$$\mathrm{kron}(\mathbf{A}, \mathbf{B}) = \begin{bmatrix} a_{00} \otimes \mathbf{B} & \cdots & a_{0,n-1} \otimes \mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} \otimes \mathbf{B} & \cdots & a_{m-1,n-1} \otimes \mathbf{B} \end{bmatrix}$$

The $\otimes$ operator is defined by the `op` parameter. It is applied in an element-wise fashion (like `GrB_eWiseMult`), where the pattern of the submatrix $a_{ij} \otimes \mathbf{B}$ is the same as the pattern of $\mathbf{B}$ if a_{ij} is an entry in the matrix $\mathbf{A}$, or empty otherwise. The input matrices `A` and `B` can be of any dimension, and both matrices may be transposed first via the descriptor, `desc`. Entries in `A` and `B` are typecast into the input types of the `op`. The matrix `T=kron(A,B)` has the same type as the `ztype` of the binary operator, `op`. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

The operator `op` may be a `GrB_BinaryOp`, a `GrB_Monoid`, or a `GrB_Semiring`. In the latter case, the multiplicative operator of the semiring is used.

11 Printing GraphBLAS objects

The ten different objects handled by SuiteSparse:GraphBLAS are all opaque, although nearly all of their contents can be extracted via methods such as `GrB_Matrix_extractTuples`, `GrB_Matrix_extractElement`, `GxB_Matrix_type`, and so on. The GraphBLAS C API has no mechanism for printing all the contents of GraphBLAS objects, but this is helpful for debugging. Ten type-specific methods and two type-generic methods are provided:

`GxB_Type_fprint`	print and check a `GrB_Type`
`GxB_UnaryOp_fprint`	print and check a `GrB_UnaryOp`
`GxB_BinaryOp_fprint`	print and check a `GrB_BinaryOp`
`GxB_IndexUnaryOP_fprint`	print and check a `GrB_IndexUnaryOp`
`GxB_Monoid_fprint`	print and check a `GrB_Monoid`
`GxB_Semiring_fprint`	print and check a `GrB_Semiring`
`GxB_Descriptor_fprint`	print and check a `GrB_Descriptor`
`GxB_Matrix_fprint`	print and check a `GrB_Matrix`
`GxB_Vector_fprint`	print and check a `GrB_Vector`
`GxB_Scalar_fprint`	print and check a `GrB_Scalar`
`GxB_fprint`	print/check any object to a file
`GxB_print`	print/check any object to `stdout`

These methods do not modify the status of any object, and thus they cannot return an error string for use by `GrB_error`.

If a matrix or vector has not been completed, the pending computations are guaranteed to *not* be performed. The reason is simple. It is possible for a bug in the user application (such as accessing memory outside the bounds of an array) to mangle the internal content of a GraphBLAS object, and the `GxB_*print` methods can be helpful tools to track down this bug. If `GxB_*print` attempted to complete any computations prior to printing or checking the contents of the matrix or vector, then further errors could occur, including a segfault.

By contrast, GraphBLAS methods and operations that return values into user-provided arrays or variables might finish pending operations before the return these values, and this would change their state. Since they do not change the state of any object, the `GxB_*print` methods provide a useful alternative for debugging, and for a quick understanding of what GraphBLAS is computing while developing a user application.

Each of the methods has a parameter of type `GxB_Print_Level` that specifies the amount to print:

```
typedef enum
{
    GxB_SILENT = 0,     // nothing is printed, just check the object
    GxB_SUMMARY = 1,    // print a terse summary
    GxB_SHORT = 2,      // short description, about 30 entries of a matrix
    GxB_COMPLETE = 3,   // print the entire contents of the object
    GxB_SHORT_VERBOSE = 4,    // GxB_SHORT but with "%.15g" for doubles
```

```
    GxB_COMPLETE_VERBOSE = 5  // GxB_COMPLETE but with "%.15g" for doubles
}
GxB_Print_Level ;
```

The ten type-specific functions include an additional argument, the `name` string. The `name` is printed at the beginning of the display (assuming the print level is not `GxB_SILENT`) so that the object can be more easily identified in the output. For the type-generic methods `GxB_fprint` and `GxB_print`, the `name` string is the variable name of the object itself.

If the file `f` is `NULL`, `stdout` is used. If `name` is `NULL`, it is treated as the empty string. These are not error conditions.

The methods check their input objects carefully and extensively, even when `pr` is equal to `GxB_SILENT`. The following error codes can be returned:

- `GrB_SUCCESS`: object is valid
- `GrB_UNINITIALIZED_OBJECT`: object is not initialized
- `GrB_INVALID_OBJECT`: object is not valid
- `GrB_NULL_POINTER`: object is a NULL pointer
- `GrB_INVALID_VALUE`: `fprintf` returned an I/O error.

The content of any GraphBLAS object is opaque, and subject to change. As a result, the exact content and format of what is printed is implementation-dependent, and will change from version to version of SuiteSparse:GraphBLAS. Do not attempt to rely on the exact content or format by trying to parse the resulting output via another program. The intent of these functions is to produce a report of an object for visual inspection. If the user application needs to extract content from a GraphBLAS matrix or vector, use `GrB_*_extractTuples` or the import/export methods instead.

GraphBLAS matrices and vectors are zero-based, where indices of an n-by-n matrix are in the range 0 to $n-1$. However, Octave, MATLAB, and Julia prefer to print their matrices and vectors as one-based. To enable 1-based printing, use `GxB_set (GxB_PRINT_1BASED, true)`. Printing is done as zero-based by default.

11.1 GxB_fprint: Print a GraphBLAS object to a file

```
GrB_Info GxB_fprint                 // print and check a GraphBLAS object
(
    GrB_<objecttype> object,        // object to print and check
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

The `GxB_fprint` function prints the contents of any of the ten GraphBLAS objects to the file `f`. If `f` is `NULL`, the results are printed to `stdout`. For example, to print the entire contents of a matrix `A` to the file `f`, use `GxB_fprint (A, GxB_COMPLETE, f)`.

11.2 GxB_print: Print a GraphBLAS object to stdout

```
GrB_Info GxB_print                  // print and check a GrB_Vector
(
    GrB_<objecttype> object,        // object to print and check
    GxB_Print_Level pr              // print level
) ;
```

`GxB_print` is the same as `GxB_fprint`, except that it prints the contents of the object to `stdout` instead of a file `f`. For example, to print the entire contents of a matrix `A`, use `GxB_print (A, GxB_COMPLETE)`.

11.3 GxB_Type_fprint: Print a GrB_Type

```
GrB_Info GxB_Type_fprint            // print and check a GrB_Type
(
    GrB_Type type,                  // object to print and check
    const char *name,               // name of the object
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

For example, `GxB_Type_fprint (GrB_BOOL, "boolean type", GxB_COMPLETE, f)` prints the contents of the `GrB_BOOL` object to the file `f`.

11.4 GxB_UnaryOp_fprint: Print a GrB_UnaryOp

```
GrB_Info GxB_UnaryOp_fprint         // print and check a GrB_UnaryOp
(
    GrB_UnaryOp unaryop,            // object to print and check
    const char *name,               // name of the object
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

For example, `GxB_UnaryOp_fprint (GrB_LNOT, "not", GxB_COMPLETE, f)` prints the `GrB_LNOT` unary operator to the file `f`.

11.5 GxB_BinaryOp_fprint: Print a GrB_BinaryOp

```
GrB_Info GxB_BinaryOp_fprint        // print and check a GrB_BinaryOp
(
    GrB_BinaryOp binaryop,          // object to print and check
    const char *name,               // name of the object
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

For example, `GxB_BinaryOp_fprint (GrB_PLUS_FP64, "plus", GxB_COMPLETE, f)` prints the `GrB_PLUS_FP64` binary operator to the file `f`.

11.6 GxB_IndexUnaryOp_fprint: Print a GrB_IndexUnaryOp

```
GrB_Info GxB_IndexUnaryOp_fprint    // print and check a GrB_IndexUnaryOp
(
    GrB_IndexUnaryOp op,            // object to print and check
    const char *name,               // name of the object
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

For example, `GrB_IndexUnaryOp_fprint (GrB_TRIL, "tril", GxB_COMPLETE, f)` prints the `GrB_TRIL` index-unary operator to the file `f`.

11.7 GxB_Monoid_fprint: Print a GrB_Monoid

```
GrB_Info GxB_Monoid_fprint          // print and check a GrB_Monoid
(
    GrB_Monoid monoid,              // object to print and check
    const char *name,               // name of the object
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

For example, `GxB_Monoid_fprint (GxB_PLUS_FP64_MONOID, "plus monoid", GxB_COMPLETE, f)` prints the predefined `GxB_PLUS_FP64_MONOID` (based on the binary operator `GrB_PLUS_FP64`) to the file `f`.

11.8 GxB_Semiring_fprint: Print a GrB_Semiring

```
GrB_Info GxB_Semiring_fprint        // print and check a GrB_Semiring
(
    GrB_Semiring semiring,          // object to print and check
    const char *name,               // name of the object
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

For example, `GxB_Semiring_fprint (GxB_PLUS_TIMES_FP64, "standard", GxB_COMPLETE, f)` prints the predefined `GxB_PLUS_TIMES_FP64` semiring to the file `f`.

11.9 GxB_Descriptor_fprint: Print a GrB_Descriptor

```
GrB_Info GxB_Descriptor_fprint      // print and check a GrB_Descriptor
(
    GrB_Descriptor descriptor,      // object to print and check
    const char *name,               // name of the object
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

For example, `GxB_Descriptor_fprint (d, "descriptor", GxB_COMPLETE, f)` prints the descriptor `d` to the file `f`.

11.10 GxB_Matrix_fprint: Print a GrB_Matrix

```
GrB_Info GxB_Matrix_fprint          // print and check a GrB_Matrix
(
    GrB_Matrix A,                   // object to print and check
    const char *name,               // name of the object
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

For example, `GxB_Matrix_fprint (A, "my matrix", GxB_SHORT, f)` prints about 30 entries from the matrix `A` to the file `f`.

11.11 GxB_Vector_fprint: Print a GrB_Vector

```
GrB_Info GxB_Vector_fprint          // print and check a GrB_Vector
(
    GrB_Vector v,                   // object to print and check
    const char *name,               // name of the object
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

For example, `GxB_Vector_fprint (v, "my vector", GxB_SHORT, f)` prints about 30 entries from the vector `v` to the file `f`.

11.12 GxB_Scalar_fprint: Print a GrB_Scalar

```
GrB_Info GxB_Scalar_fprint          // print and check a GrB_Scalar
(
    GrB_Scalar s,                   // object to print and check
    const char *name,               // name of the object
    GxB_Print_Level pr,             // print level
    FILE *f                         // file for output
) ;
```

For example, `GxB_Scalar_fprint (s, "my scalar", GxB_SHORT, f)` prints a short description of the scalar `s` to the file `f`.

11.13 Performance and portability considerations

Even when the print level is `GxB_SILENT`, these methods extensively check the contents of the objects passed to them, which can take some time. They should be considered debugging tools only, not for final use in production.

The return value of the `GxB_*print` methods can be relied upon, but the output to the file (or `stdout`) can change from version to version. If these methods are eventually added to the GraphBLAS C API Specification, a conforming implementation might never print anything at all, regardless of the `pr` value. This may be essential if the GraphBLAS library is installed in a dedicated device, with no file output, for example.

Some implementations may wish to print nothing at all if the matrix is not yet completed, or just an indication that the matrix has pending operations and cannot be printed, when non-blocking mode is employed. In this case, use `GrB_Matrix_wait`, `GrB_Vector_wait`, or `GxB_Scalar_wait` to finish all pending computations first. If a matrix or vector has pending operations, SuiteSparse:GraphBLAS prints a list of the *pending tuples*, which are the entries not yet inserted into the primary data structure. It can also print out entries that remain in the data structure but are awaiting deletion; these are called *zombies* in the output report.

Most of the rest of the report is self-explanatory.

12 Matrix and Vector iterators

The `GxB_Iterator` is an object that allows user applications to iterate over the entries of a matrix or vector, one entry at a time. Iteration can be done in a linear manner (analogous to reading a file one entry at a time, from start to finish), or in a random-access pattern (analogous to the `fseek` method for repositioning the access to file to a different position).

Multiple iterators can be used on a single matrix or vector, even in parallel by multiple user threads. While a matrix or vector is being used with an iterator, the matrix or vector must not be modified. Doing so will lead to undefined results.

Since accessing a matrix or vector via an iterator requires many calls to the iterator methods, they must be very fast. Error checking is skipped, except for the methods that create, attach, or free an iterator. Methods that advance an iterator or that access values or indices from a matrix or vector do not return error conditions. Instead, they have well-defined preconditions that must be met (and which should be checked by the user application). If those preconditions are not met, results are undefined.

The iterator methods are implemented in SuiteSparse:GraphBLAS as both macros (via `#define`) and as functions of the same name that appear in the compiled `libgraphblas.so` library. This requires that the opaque contents of the iterator object be defined in `GraphBLAS.h` itself. The user application must not access these contents directly, but can only do so safely via the iterator methods provided by SuiteSparse:GraphBLAS.

The iterator object can be used in one of four sets of methods, for four different access patterns:

1. *row iterator*: iterates across the rows of a matrix, and then within each row to access the entries in a given row. Accessing all the entries of a matrix using a row iterator requires an outer loop (for the rows) and an inner loop (for the entries in each row). A matrix can be accessed via a row iterator only if its format (determined by `GxB_get (A, GxB_FORMAT, &fmt)`) is by-row (that is, `GxB_BY_ROW`). See Section 8.

2. *column iterator*: iterates across the columns of a matrix, and then within each column to access the entries in a given column. Accessing all the entries of a matrix using a column iterator requires an outer loop (for the columns) and an inner loop (for the entries in each column). A matrix can be accessed via a column iterator only if its format (determined by `GxB_get (A, GxB_FORMAT, &fmt)`) is by-column (that is, `GxB_BY_COL`). See Section 8.

3. *entry iterator*: iterates across the entries of a matrix. Accessing all the entries of a matrix using an entry iterator requires just a single loop. Any matrix can be accessed with an entry iterator.

4. *vector iterator*: iterates across the entries of a vector. Accessing all the entries of a vector using a vector iterator requires just a single loop. Any vector can be accessed with a vector iterator.

12.1 Creating and destroying an iterator

The process for using an iterator starts with the creation of an iterator, with `GxB_Iterator_new`. This method creates an `iterator` object but does not *attach* it to any specific matrix or vector:

```
GxB_Iterator iterator ;
GxB_Iterator_new (&iterator) ;
```

When finished, the `iterator` is freed with either of these methods:

```
GrB_free (&iterator) ;
GxB_Iterator_free (&iterator) ;
```

12.2 Attaching an iterator to a matrix or vector

This new `iterator` object can be *attached* to any matrix or vector, and used as a row, column, or entry iterator for any matrix, or as an iterator for any vector. The `iterator` can be used in any of these methods before it is freed, but with just one access method at a time.

Once it is created, the `iterator` must be attached to a matrix or vector. This process also selects the method by which the `iterator` will be used for a matrix. Each of the four `GxB_*Iterator_attach` methods returns a `GrB_Info` result. The descriptor `desc` in the examples below is used only to control the number of threads used for the internal call to `GrB_wait`, if the matrix `A` or vector `v` has pending operations.

1. *row iterator*:

   ```
   GrB_Info info = GxB_rowIterator_attach (iterator, A, desc) ;
   ```

2. *column iterator*:

   ```
   GrB_Info info = GxB_colIterator_attach (iterator, A, desc) ;
   ```

3. *entry iterator*:

   ```
   GrB_Info info = GxB_Matrix_Iterator_attach (iterator, A, desc) ;
   ```

4. *vector iterator*:

   ```
   GrB_Info info = GxB_Vector_Iterator_attach (iterator, v, desc) ;
   ```

On input to `GxB_*Iterator_attach`, the `iterator` must already exist, having been created by `GxB_Iterator_new`. If the `iterator` is already attached to a matrix or vector, it is detached and then attached to the given matrix `A` or vector `v`.

The return values for row/column methods are:

- `GrB_SUCCESS`: if the `iterator` is successfully attached to the matrix `A`.

- `GrB_NULL_POINTER`: if the `iterator` or `A` are NULL.
- `GrB_INVALID_OBJECT`: if the matrix `A` is invalid.
- `GrB_NOT_IMPLEMENTED`: if the matrix `A` cannot be iterated in the requested access method (row iterators require the matrix to be held by-row, and column iterators require the matrix to be held by-column).
- `GrB_OUT_OF_MEMORY`: if the method runs out of memory.

The other two methods (entry iterator for matrices, or the vector iterator) return the same error codes, except that they do not return `GrB_NOT_IMPLEMENTED`.

12.3 Seeking to an arbitrary position

Attaching the `iterator` to a matrix or vector does not define a specific position for the `iterator`. To use the `iterator`, a single call to the corresponding *seek* method is required. These `GxB*_Iterator_*seek*` methods may also be used later on to change the position of the iterator arbitrarily.

1. *row iterator*:

```
GrB_Info info = GxB_rowIterator_seekRow (iterator, row) ;
GrB_Index kount = GxB_rowIterator_kount (iterator) ;
GrB_Info info = GxB_rowIterator_kseek (iterator, k) ;
```

   These methods move a row iterator to a specific row, defined in one of two ways: (1) the row index itself (in range 0 to `nrows`-1), or (2) by specifying `k`, which moves the iterator to the `k`th *explicit* row (in the range 0 to `kount`-1). For sparse, bitmap, or full matrices, these two methods are identical. For hypersparse matrices, not all rows are present in the data structure; these *implicit* rows are skipped and not included in the `kount`. Implicit rows contain no entries. The `GxB_rowIterator_kount` method returns the `kount` of the matrix, where `kount` is equal to `nrows` for sparse, bitmap, and matrices, and `kount` $\leq$ `nrows` for hypersparse matrices. All three methods listed above can be used for any row iterator.

   The `GxB_rowIterator_*seek*` methods return `GrB_SUCCESS` if the iterator has been moved to a row that contains at least one entry, `GrB_NO_VALUE` if the row has no entries, or `GxB_EXHAUSTED` if the row is out of bounds (`row` $\geq$ `nrows` or if `k` $\geq$ `kount`). None of these return conditions are errors; they are all informational.

   For sparse, bitmap, and full matrices, `GxB_rowIterator_seekRow` always moves to the given row. For hypersparse matrices, if the requested row is implicit, the iterator is moved to the first explicit row following it. If no such row exists, the iterator is exhausted and `GxB_EXHAUSTED` is returned. The `GxB_rowIterator_kseek` method always moves to the `k`th explicit row, for any matrix. Use `GxB_rowIterator_getRowIndex`, described below, to determine the row index of the current position.

   Precondition: on input, the `iterator` must have been successfully attached to a matrix via a prior call to `GxB_rowIterator_attach`. Results are undefined if this precondition is not met.

2. *column iterator*:

```
GrB_Info info = GxB_colIterator_seekCol (iterator, col) ;
GrB_Index kount = GxB_colIterator_kount (iterator) ;
GrB_Info info = GxB_colIterator_kseek (iterator, k) ;
```

These methods move a column iterator to a specific column, defined in one of two ways: (1) the column index itself (in range 0 to `ncols`-1), or (2) by specifying `k`, which moves the iterator to the `k`th *explicit* column (in the range 0 to `kount`-1). For sparse, bitmap, or full matrices, these two methods are identical. For hypersparse matrices, not all columns are present in the data structure; these *implicit* columns are skipped and not included in the `kount`. Implicit columns contain no entries. The `GxB_colIterator_kount` method returns the `kount` of the matrix, where `kount` is equal to `ncols` for sparse, bitmap, and matrices, and `kount` $\leq$ `ncols` for hypersparse matrices. All three methods listed above can be used for any column iterator.

The `GxB_colIterator_*seek*` methods return `GrB_SUCCESS` if the iterator has been moved to a column that contains at least one entry, `GrB_NO_VALUE` if the column has no entries, or `GxB_EXHAUSTED` if the column is out of bounds (`col` $\geq$ `ncols` or `k` $\geq$ `kount`). None of these return conditions are errors; they are all informational.

For sparse, bitmap, and full matrices, `GxB_colIterator_seekCol` always moves to the given column. For hypersparse matrices, if the requested column is implicit, the iterator is moved to the first explicit column following it. If no such column exists, the iterator is exhausted and `GxB_EXHAUSTED` is returned. The `GxB_colIterator_kseek` method always moves to the `k`th explicit column, for any matrix. Use `GxB_colIterator_getColIndex`, described below, to determine the column index of the current position.

Precondition: on input, the `iterator` must have been successfully attached to a matrix via a prior call to `GxB_colIterator_attach`. Results are undefined if this precondition is not met.

3. *entry iterator*:

```
GrB_Info info = GxB_Matrix_Iterator_seek (iterator, p) ;
GrB_Index pmax = GxB_Matrix_Iterator_getpmax (iterator) ;
GrB_Index p = GxB_Matrix_Iterator_getp (iterator);
```

The `GxB_Matrix_Iterator_seek` method moves the `iterator` to the given position `p`, which is in the range 0 to `pmax`-1, where the value of `pmax` is obtained from `GxB_Matrix_Iterator_getpmax`. For sparse, hypersparse, and full matrices, `pmax` is the same as `nvals` returned by `GrB_Matrix_nvals`. For bitmap matrices, `pmax` is equal to `nrows*ncols`. If `p` $\geq$ `pmax`, the iterator is exhausted and `GxB_EXHAUSTED` is returned. Otherwise, `GrB_SUCCESS` is returned.

All entries in the matrix are given an ordinal position, `p`. Seeking to position `p` will either move the `iterator` to that particular position, or to the next higher position containing an entry if there is entry at position `p`. The latter case only occurs

for bitmap matrices. Use `GxB_Matrix_Iterator_getp` to determine the current position of the iterator.

Precondition: on input, the `iterator` must have been successfully attached to a matrix via a prior call to `GxB_Matrix_Iterator_attach`. Results are undefined if this precondition is not met.

4. *vector iterator*:

```
GrB_Info info = GxB_Vector_Iterator_seek (iterator, p) ;
GrB_Index pmax = GxB_Vector_Iterator_getpmax (iterator) ;
GrB_Index p = GxB_Vector_Iterator_getp (iterator);
```

The `GxB_Vector_Iterator_seek` method is identical to the entry iterator of a matrix, but applied to a `GrB_Vector` instead.

Precondition: on input, the `iterator` must have been successfully attached to a vector via a prior call to `GxB_Vector_Iterator_attach`. Results are undefined if this precondition is not met.

12.4 Advancing to the next position

For best performance, the *seek* methods described above should be used with care, since some of them require $O(\log n)$ time. The fastest method for changing the position of the iterator is the corresponding *next* method, described below for each iterator:

1. *row iterator*: To move to the next row.

```
GrB_Info info = GxB_rowIterator_nextRow (iterator) ;
```

The row iterator is a 2-dimensional iterator, requiring an outer loop and an inner loop. The outer loop iterates over the rows of the matrix, using `GxB_rowIterator_nextRow` to move to the next row. If the matrix is hypersparse, the next row is always an explicit row; implicit rows are skipped. The return conditions are identical to `GxB_rowIterator_seekRow`.

Preconditions: on input, the row iterator must already be attached to a matrix via a prior call to `GxB_rowIterator_attach`, and the `iterator` must be at a specific row, via a prior call to `GxB_rowIterator_*seek*` or `GxB_rowIterator_nextRow`. Results are undefined if these conditions are not met.

2. *row iterator*: To move to the next entry within a row.

```
GrB_Info info = GxB_rowIterator_nextCol (iterator) ;
```

The row iterator is moved to the next entry in the current row. The method returns `GrB_NO_VALUE` if the end of the row is reached. The iterator does not move to the next row in this case. The method returns `GrB_SUCCESS` if the iterator has been moved to a specific entry in the current row.

Preconditions: the same as `GxB_rowIterator_nextRow`.

3. *column iterator*: To move to the next column

```
GrB_Info info = GxB_colIterator_nextCol (iterator) ;
```

The column iterator is a 2-dimensional iterator, requiring an outer loop and an inner loop. The outer loop iterates over the columns of the matrix, using `GxB_colIterator_nextCol` to move to the next column. If the matrix is hypersparse, the next column is always an explicit column; implicit columns are skipped. The return conditions are identical to `GxB_colIterator_seekCol`.

Preconditions: on input, the column iterator must already be attached to a matrix via a prior call to `GxB_colIterator_attach`, and the `iterator` must be at a specific column, via a prior call to `GxB_colIterator_*seek*` or `GxB_colIterator_nextCol`. Results are undefined if these conditions are not met.

4. *column iterator*: To move to the next entry within a column.

```
GrB_Info info = GxB_colIterator_nextRow (iterator) ;
```

The column iterator is moved to the next entry in the current column. The method returns `GrB_NO_VALUE` if the end of the column is reached. The iterator does not move to the next column in this case. The method returns `GrB_SUCCESS` if the iterator has been moved to a specific entry in the current column.

Preconditions: the same as `GxB_colIterator_nextCol`.

5. *entry iterator*: To move to the next entry.

```
GrB_Info info = GxB_Matrix_Iterator_next (iterator) ;
```

This method moves an iterator to the next entry of a matrix. It returns `GrB_SUCCESS` if the iterator is at an entry that exists in the matrix, or `GrB_EXHAUSTED` otherwise.

Preconditions: on input, the entry iterator must be already attached to a matrix via `GxB_Matrix_Iterator_attach`, and the position of the iterator must also have been defined by a prior call to `GxB_Matrix_Iterator_seek` or `GxB_Matrix_Iterator_next`. Results are undefined if these conditions are not met.

6. *vector iterator*: To move to the next entry.

```
GrB_Info info = GxB_Vector_Iterator_next (iterator) ;
```

This method moves an iterator to the next entry of a vector. It returns `GrB_SUCCESS` if the iterator is at an entry that exists in the vector, or `GrB_EXHAUSTED` otherwise.

Preconditions: on input, the iterator must be already attached to a vector via `GxB_Vector_Iterator_attach`, and the position of the iterator must also have been defined by a prior call to `GxB_Vector_Iterator_seek` or `GxB_Vector_Iterator_next`. Results are undefined if these conditions are not met.

12.5 Accessing the indices of the current entry

Once the iterator is attached to a matrix or vector, and is placed in position at an entry in the matrix or vector, the indices and value of this entry can be obtained. The methods for accessing the value of the entry are described in Section 12.6. Accessing the indices is performed with four different sets of methods, depending on which access pattern is in use, described below:

1. *row iterator*: To get the current row index.

   ```
   GrB_Index i = GxB_rowIterator_getRowIndex (iterator) ;
   ```

   The method returns `nrows(A)` if the iterator is exhausted, or the current row index `i` otherwise. There need not be any entry in the current row. Zero is returned if the iterator is attached to the matrix but `GxB_rowIterator_*seek*` has not been called, but this does not mean the iterator is positioned at row zero.

   Preconditions: on input, the iterator must be already successfully attached to matrix as a row iterator via `GxB_rowIterator_attach`. Results are undefined if this condition is not met.

2. *row iterator*: To get the current column index.

   ```
   GrB_Index j = GxB_rowIterator_getColIndex (iterator) ;
   ```

   Preconditions: on input, the iterator must be already successfully attached to matrix as a row iterator via `GxB_rowIterator_attach`, and in addition, the row iterator must be positioned at a valid entry present in the matrix. That is, the last call to `GxB_rowIterator_*seek*` or `GxB_rowIterator_*next*`, must have returned `GrB_SUCCESS`. Results are undefined if these conditions are not met.

3. *column iterator*: To get the current column index.

   ```
   GrB_Index j = GxB_colIterator_getColIndex (iterator) ;
   ```

   The method returns `ncols(A)` if the iterator is exhausted, or the current column index `j` otherwise. There need not be any entry in the current column. Zero is returned if the iterator is attached to the matrix but `GxB_colIterator_*seek*` has not been called, but this does not mean the iterator is positioned at column zero.

   Precondition: on input, the iterator must be already successfully attached to matrix as a column iterator via `GxB_colIterator_attach`. Results are undefined if this condition is not met.

4. *column iterator*: To get the current row index.

   ```
   GrB_Index i = GxB_colIterator_getRowIndex (iterator) ;
   ```

Preconditions: on input, the iterator must be already successfully attached to matrix as a column iterator via `GxB_colIterator_attach`, and in addition, the column iterator must be positioned at a valid entry present in the matrix. That is, the last call to `GxB_colIterator_*seek*` or `GxB_colIterator_*next*`, must have returned `GrB_SUCCESS`. Results are undefined if these conditions are not met.

5. *entry iterator*: To get the current row and column index.

```
GrB_Index i, j ;
GxB_Matrix_Iterator_getIndex (iterator, &i, &j) ;
```

Returns the row and column index of the current entry.

Preconditions: on input, the entry iterator must be already attached to a matrix via `GxB_Matrix_Iterator_attach`, and the position of the iterator must also have been defined by a prior call to `GxB_Matrix_Iterator_seek` or `GxB_Matrix_Iterator_next`, with a return value of `GrB_SUCCESS`. Results are undefined if these conditions are not met.

6. *vector iterator*: To get the current index.

```
GrB_Index i = GxB_Vector_Iterator_getIndex (iterator) ;
```

Returns the index of the current entry.

Preconditions: on input, the entry iterator must be already attached to a matrix via `GxB_Vector_Iterator_attach`, and the position of the iterator must also have been defined by a prior call to `GxB_Vector_Iterator_seek` or `GxB_Vector_Iterator_next`, with a return value of `GrB_SUCCESS`. Results are undefined if these conditions are not met.

12.6 Accessing the value of the current entry

So far, all methods that create or use an iterator have been split into four sets of methods, for the row, column, or entry iterators attached to a matrix, or for a vector iterator. Accessing the value is different. All four iterators use the same set of methods to access the value of their current entry. These methods return the value of the current entry at the position determined by the iterator. The return value can of course be typecasted using standard C syntax once the value is returned to the caller.

Preconditions: on input, the prior call to `GxB_*Iterator_*seek*`, or `GxB_*Iterator_*next*` must have returned `GrB_SUCCESS`, indicating that the iterator is at a valid current entry for either a matrix or vector. No typecasting is permitted, in the sense that the method name must match the type of the matrix or vector. Results are undefined if these conditions are not met.

```
// for built-in types:
bool       value = GxB_Iterator_get_BOOL (iterator) ;
int8_t     value = GxB_Iterator_get_INT8 (iterator) ;
int16_t    value = GxB_Iterator_get_INT16 (iterator) ;
```

```
int32_t    value = GxB_Iterator_get_INT32 (iterator) ;
int64_t    value = GxB_Iterator_get_INT64 (iterator) ;
uint8_t    value = GxB_Iterator_get_UINT8 (iterator) ;
uint16_t   value = GxB_Iterator_get_UINT16 (iterator) ;
uint32_t   value = GxB_Iterator_get_UINT32 (iterator) ;
uint64_t   value = GxB_Iterator_get_UINT64 (iterator) ;
float      value = GxB_Iterator_get_FP32 (iterator) ;
double     value = GxB_Iterator_get_FP64 (iterator) ;
GxB_FC32_t value = GxB_Iterator_get_FC32 (iterator) ;
GxB_FC64_t value = GxB_Iterator_get_FC64 (iterator) ;

// for user-defined types:
<type> value ;
GxB_Iterator_get_UDT (iterator, (void *) &value) ;
```

12.7 Example: row iterator for a matrix

The following example uses a row iterator to access all of the entries in a matrix `A` of type `GrB_FP64`. Note the inner and outer loops. The outer loop iterates over all rows of the matrix. The inner loop iterates over all entries in the row `i`. This access pattern requires the matrix to be held by-row, but otherwise it works for any matrix. If the matrix is held by-column, then use the column iterator methods instead.

```
// create an iterator
GxB_Iterator iterator ;
GxB_Iterator_new (&iterator) ;
// attach it to the matrix A, known to be type GrB_FP64
GrB_Info info = GxB_rowIterator_attach (iterator, A, NULL) ;
if (info < 0) { handle the failure ... }
// seek to A(0,:)
info = GxB_rowIterator_seekRow (iterator, 0) ;
while (info != GxB_EXHAUSTED)
{
    // iterate over entries in A(i,:)
    GrB_Index i = GxB_rowIterator_getRowIndex (iterator) ;
    while (info == GrB_SUCCESS)
    {
        // get the entry A(i,j)
        GrB_Index j = GxB_rowIterator_getColIndex (iterator) ;
        double  aij = GxB_Iterator_get_FP64 (iterator) ;
        // move to the next entry in A(i,:)
        info = GxB_rowIterator_nextCol (iterator) ;
    }
    // move to the next row, A(i+1,:), or a subsequent one if i+1 is implicit
    info = GxB_rowIterator_nextRow (iterator) ;
}
GrB_free (&iterator) ;
```

12.8 Example: column iterator for a matrix

The column iterator is analgous to the row iterator.

The following example uses a column iterator to access all of the entries in a matrix `A` of type `GrB_FP64`. The outer loop iterates over all columns of the matrix. The inner loop iterates over all entries in the column `j`. This access pattern requires the matrix to be held by-column, but otherwise it works for any matrix. If the matrix is held by-row, then use the row iterator methods instead.

```
// create an iterator
GxB_Iterator iterator ;
GxB_Iterator_new (&iterator) ;
// attach it to the matrix A, known to be type GrB_FP64
GrB_Info info = GxB_colIterator_attach (iterator, A, NULL) ;
// seek to A(:,0)
info = GxB_colIterator_seekCol (iterator, 0) ;
while (info != GxB_EXHAUSTED)
{
    // iterate over entries in A(:,j)
    GrB_Index j = GxB_colIterator_getColIndex (iterator) ;
    while (info == GrB_SUCCESS)
    {
        // get the entry A(i,j)
        GrB_Index i = GxB_colIterator_getRowIndex (iterator) ;
        double  aij = GxB_Iterator_get_FP64 (iterator) ;
        // move to the next entry in A(:,j)
        info = GxB_colIterator_nextRow (iterator) ;
        OK (info) ;
    }
    // move to the next column, A(:,j+1), or a subsequent one if j+1 is implicit
    info = GxB_colIterator_nextCol (iterator) ;
}
GrB_free (&iterator) ;
```

12.9 Example: entry iterator for a matrix

The entry iterator allows for a simpler access pattern, with a single loop, but using a row or column iterator is faster. The method works for any matrix.

```
// create an iterator
GxB_Iterator iterator ;
GxB_Iterator_new (&iterator) ;
// attach it to the matrix A, known to be type GrB_FP64
GrB_Info info = GxB_Matrix_Iterator_attach (iterator, A, NULL) ;
if (info < 0) { handle the failure ... }
// seek to the first entry
info = GxB_Matrix_Iterator_seek (iterator, 0) ;
while (info != GxB_EXHAUSTED)
{
    // get the entry A(i,j)
    GrB_Index i, j ;
    GxB_Matrix_Iterator_getIndex (iterator, &i, &j) ;
    double aij = GxB_Iterator_get_FP64 (iterator) ;
    // move to the next entry in A
    info = GxB_Matrix_Iterator_next (iterator) ;
}
GrB_free (&iterator) ;
```

12.10 Example: vector iterator

A vector iterator is used much like an entry iterator for a matrix.

```
// create an iterator
GxB_Iterator iterator ;
GxB_Iterator_new (&iterator) ;
// attach it to the vector v, known to be type GrB_FP64
GrB_Info info = GxB_Vector_Iterator_attach (iterator, v, NULL) ;
if (info < 0) { handle the failure ... }
// seek to the first entry
info = GxB_Vector_Iterator_seek (iterator, 0) ;
while (info != GxB_EXHAUSTED)
{
    // get the entry v(i)
    GrB_Index i = GxB_Vector_Iterator_getIndex (iterator) ;
    double vi = GxB_Iterator_get_FP64 (iterator) ;
    // move to the next entry in v
    info = GxB_Vector_Iterator_next (iterator) ;
}
GrB_free (&iterator) ;
```

12.11 Performance

I have benchmarked the performance of the row and column iterators to compute `y=0` and then `y+=A*x` where `y` is a dense vector and `A` is a sparse matrix, using a single thread. The row and column iterators are very fast, sometimes only 1% slower than calling `GrB_mxv` to compute the same thing (also assuming a single thread), for large problems. For sparse matrices that average just 1 or 2 entries per row, the row iterator can be about 30% slower than `GrB_mxv`, likely because of the slightly higher complexity of moving from one row to the next using these methods.

It is possible to split up the problem for multiple user threads, each with its own iterator. Given the low overhead of the row and column iterator for a single thread, this should be very fast. Care must be taken to ensure a good load balance. Simply spliting up the rows of a matrix and giving the same number of rows to each user thread can result in imbalanced work. This is handled internally in `GrB_*` methods, but enabling parallelism when using iterators is the responsibility of the user application.

The entry iterators are easier to use but harder to implement. The methods must internally fuse both inner and outer loops so that the user application can use a single loop. As a result, the computation `y+=A*x` can be up to 4x slower (about 2x typical) than when using `GrB_mxv` with a single thread.

To obtain the best performace possible, many of the iterator methods are implemented as macros in `GraphBLAS.h`. Using macros is the default, giving typical C and C++ applications access to the fastest methods possible.

To ensure access to these methods when not using the macros, these methods are also defined as regular functions that appear in the compiled `libgraphblas.so` library with the same name as the macros. Applications that cannot use the macro versions can `#undef` the macros after the `#include <GraphBLAS.h>` statement, and then they would access the regular compiled functions in `libgraphblas.so`. This non-macro approach is not the default, and the iterator methods may be slightly slower.

13 Iso-Valued Matrices and Vectors

The GraphBLAS C API states that the entries in all `GrB_Matrix` and `GrB_Vector` objects have a numerical value, with either a built-in or user-defined type. Representing an unweighted graph requires a value to be placed on each edge, typically $a_{ij} = 1$. Adding a structure-only data type would not mix well with the rest of GraphBLAS, where all operators, monoids, and semirings need to operate on a value, of some data type. And yet unweighted graphs are very important in graph algorithms.

The solution is simple, and exploiting it in SuiteSparse:GraphBLAS requires nearly no extensions to the GraphBLAS C API. SuiteSparse:GraphBLAS can often detect when the user application is creating a matrix or vector where all entries in the sparsity pattern take on the same numerical value.

For example, $\mathbf{C}\langle\mathbf{C}\rangle = 1$, when the mask is structural, sets all entries in $\mathbf{C}$ to the value 1. SuiteSparse:GraphBLAS detects this, and performs this assignment in $O(1)$ time. It stores a single copy of this "iso-value" and sets an internal flag in the opaque data structure for $\mathbf{C}$, which states that all entries in the pattern of $\mathbf{C}$ are equal to 1. This saves both time and memory and allows for the efficient representation of sparse adjacency matrices of unweighted graphs, yet does not change the C API. To the user application, it still appears that $\mathbf{C}$ has `nvals(C)` entries, all equal to 1.

Creating and operating on iso-valued matrices (or just *iso matrices* for short) is significantly faster than creating matrices with different data values. A matrix that is iso requires only $O(1)$ space for its numerical values. The sparse and hypersparse formats require an additional $O(n+e)$ or $O(e)$ integer space to hold the pattern of an n-by-n matrix `C`, respectively, and a matrix `C` in bitmap format requires $O(n^2)$ space for the bitmap. A full matrix requires no integer storage, so a matrix that is both iso and full requires only $O(1)$ space, regardless of its dimension.

The sections below a describe the methods that can be used to create iso matrices and vectors. Let a, b, and c denote the iso values of `A`, `B`, and `C`, respectively.

13.1 Using iso matrices and vectors in a graph algorithm

There are two primary useful ways to use iso-valued matrices and vectors: (1) as iso sparse/hypersparse adjacency matrices for unweighted graphs, and (2) as iso full matrices or vectors used with operations that do not need to access all of the content of the iso full matrix or vector.

In the first use case, simply create a `GrB_Matrix` with values that are all the same (those in the sparsity pattern). The `GxB_Matrix_build_Scalar` method can be used for this, since it guarantees that the time and work spent on the numerical part of the array is only $O(1)$. The method still must spend $O(e)$ or $O(e \log e)$ time on the integer arrays that represent the sparsity pattern, but the reduction in time and work on the numerical part of the matrix will improve performance.

The use of `GxB_Matrix_build_Scalar` is optional. Matrices can also be constructed with `GrB*` methods. In particular, `GrB_Matrix_build_*` can be used. It first builds a non-iso matrix and then checks if all of the values are the same, after assembling any

duplicate entries. This does not save time or memory for the construction of the matrix itself, but it will lead to savings in time and memory later on, when the matrix is used.

To ensure a matrix `C` is iso-valued, simply use `GrB_assign` to compute `C<C,struct>=1`, or assign whatever value of scalar you wish. It is essential to use a structural mask. Otherwise, it is not clear that all entries in `C` will be assigned the same value. The following code takes $O(1)$ time, and it resets the size of the numerical part of the `C` matrix to be $O(1)$ in size:

```
bool scalar = true ;
GrB_Matrix_assign (C, C, NULL, scalar, GrB_ALL, nrows, GrB_ALL, ncols,
    GrB_DESC_S) ;
```

The Octave/MATLAB analog of the code above is `C=spones(C)`.

The second case for where iso matrices and vectors are useful is to use them with operations that do not necessarily access all of their content. Suppose you have a matrix `A` of arbitrarily large dimension (say `n`-by-`n` where `n=2^60`, of type `GrB_FP64`. A matrix this large can be represented by SuiteSparse:GraphBLAS, but only in a hypersparse form.

Now, suppose you wish to compute the maximum value in each row, reducing the matrix to a vector. This can be done with `GrB_reduce`:

```
GrB_Vector_new (&v, GrB_FP64, n) ;
GrB_reduce (v, NULL, GrB_MAX_MONOID_FP64, A, NULL) ;
```

It can also be done with `GrB_mxv`, by creating an iso full vector `x`. The creation of `x` takes $O(1)$ time and memory, and the `GrB_mxv` computation takes $O(e)$ time (with modest assumptions; if `A` needs to be transposed the time would be $O(e \log e)$).

```
GrB_Vector_new (&v, GrB_FP64, n) ;
GrB_Vector_new (&x, GrB_FP64, n) ;
GrB_assign (x, NULL, NULL, 1, GrB_ALL, n, NULL) ;
GrB_mxv (v, NULL, NULL, GrB_MAX_FIRST_SEMIRING_FP64, A, x, NULL) ;
```

The above computations are identical in SuiteSparse:GraphBLAS. Internally, `GrB_reduce` creates `x` and calls `GrB_mxv`. Using `GrB_mxm` directly gives the user application additional flexibility in creating new computations that exploit the multiplicative operator in the semiring. `GrB_reduce` always uses the `FIRST` operator in its semiring, but any other binary operator can be used instead when using `GrB_mxv`.

Below is a method for computing the argmax of each row of a square matrix `A` of dimension `n` and type `GrB_FP64`. The vector `x` contains the maximum value in each row, and the vector `p` contains the zero-based column index of the maximum value in each row. If there are duplicate maximum values in each row, any one of them is selected arbitrarily using the `ANY` monoid. To select the minimum column index of the duplicate maximum values, use the `GxB_MIN_SECONDI_INT64` semiring instead (this will be slightly slower than the `ANY` monoid if there are many duplicates).

To compute the argmax of each column, use the `GrB_DESC_T0` descriptor in `GrB_mxv`, and compute `G=A*D` instead of `G=D*A` with `GrB_mxm`. See the `GrB.argmin` and `GrB.argmax` functions in the Octave/MATLAB interface for details.

```
GrB_Vector_new (&x, GrB_FP64, n) ;
GrB_Vector_new (&y, GrB_FP64, n) ;
GrB_Vector_new (&p, GrB_INT64, n) ;
// y (:) = 1, an iso full vector
GrB_assign (y, NULL, NULL, 1, GrB_ALL, n, NULL) ;
// x = max (A) where x(i) = max (A (i,:))
GrB_mxv (x, NULL, NULL, GrB_MAX_FIRST_SEMIRING_FP64, A, y, NULL) ;
// D = diag (x)
GrB_Matrix_new (&D, GrB_FP64, n, n) ;
GrB_Matrix_diag (D, x, 0) ;
// G = D*A using the ANY_EQ semiring
GrB_Matrix_new (&G, GrB_BOOL, n, n) ;
GrB_mxm (G, NULL, NULL, GxB_ANY_EQ_FP64, D, A, NULL) ;
// drop explicit zeros from G
GrB_select (G, NULL, NULL, GrB_VALUENE_BOOL, G, 0, NULL) ;
// find the position of any max entry in each row: p = G*y,
// so that p(i) = j if x(i) = A(i,j) = max (A (i,:))
GrB_mxv (p, NULL, NULL, GxB_ANY_SECONDI_INT64, G, y, NULL) ;
```

No part of the above code takes $\Omega(n)$ time or memory. The data type of the iso full vector `y` can be anything, and its iso value can be anything. It is operated on by the `FIRST` operator in the first `GrB_mxv`, and the `SECONDI` positional operator in the second `GrB_mxv`, and both operators are oblivious to the content and even the type of `y`. The semirings simply note that `y` is a full vector and compute their result according, by accessing the matrices only (`A` and `G`, respectively).

For floating-point values, `NaN` values are ignored, and treated as if they were not present in the input matrix, unless all entries in a given row are equal to `NaN`. In that case, if all entries in `A(i,:)` are equal to `NaN`, then `x(i)` is `NaN` and the entry `p(i)` is not present.

13.2 Iso matrices from matrix multiplication

Consider `GrB_mxm`, `GrB_mxv`, and `GrB_vxm`, and let `C=A*B`, where no mask is present, or `C<M>=A*B` where `C` is initially empty. If `C` is not initially empty, then these rules apply to a temporary matrix `T<M>=A*B`, which is initially empty and is then assigned to `C` via `C<M>=T`.

The iso property of `C` is determined with the following rules, where the first rule that fits defines the property and value of `C`.

- If the semiring includes a positional multiplicative operator (`GxB_FIRSTI`, `GrB_SECONDI`, and related operators), then `C` is never iso.
- Define an *iso-monoid* as a built-in monoid with the property that reducing a set of $n > 1$ identical values x returns the same value x. These are the `MIN MAX LOR LAND BOR BAND` and `ANY` monoids. All other monoids are not iso monoids: `PLUS`, `TIMES`, `LXNOR`, `EQ`, `BXOR`, `BXNOR`, and all user-defined monoids. Currently, there is

no mechanism for telling SuiteSparse:GraphBLAS that a user-defined monoid is an iso-monoid.

- If the multiplicative op is `PAIR` (same as `ONEB`), and the monoid is an iso-monoid, or the `EQ` or `TIMES` monoids, then `C` is iso with a value of 1.
- If both `B` and the monoid are iso, and the multiplicative op is `SECOND` or `ANY`, then `C` is iso with a value of b.
- If both `A` and the monoid are iso, and the multiplicative op is `FIRST` or `ANY`, then `C` is iso with a value of a.
- If `A`, `B`, and the monoid are all iso, then `C` is iso, with a value $c = f(a, b)$, where f is any multiplicative op (including user-defined, which assumes that a user-defined f has no side effects).
- If `A` and `B` are both iso and full (all entries present, regardless of the format of the matrices), then `C` is iso and full. Its iso value is computed in $O(\log(n))$ time, via a reduction of n copies of the value $t = f(a, b)$ to a scalar. The storage required to represent `C` is just $O(1)$, regardless of its dimension. Technically, the `PLUS` monoid could be computed as $c = nt$ in $O(1)$ time, but the log-time reduction works for any monoid, including user-defined ones.
- Otherwise, `C` is not iso.

13.3 Iso matrices from eWiseMult and kronecker

Consider `GrB_eWiseMult`. Let `C=A.*B`, or `C<M>=A.*B` with any mask and where `C` is initially empty, where `.*` denotes a binary operator $f(x, y)$ applied with `eWiseMult`. These rules also apply to `GrB_kronecker`.

- If the operator is positional (`GxB_FIRSTI` and related) then `C` is not iso.
- If the op is `PAIR` (same as `ONEB`), then `C` is iso with $c = 1$.
- If `B` is iso and the op is `SECOND` or `ANY`, then `C` is iso with $c = b$.
- If `A` is iso and the op is `FIRST` or `ANY`, then `C` is iso with $c = a$.
- If both `A` and `B` are iso, then `C` is iso with $c = f(a, b)$.
- Otherwise, `C` is not iso.

13.4 Iso matrices from eWiseAdd

Consider `GrB_eWiseAdd`, and also the accumulator phase of `C<M>+=T` when an accumulator operator is present. Let `C=A+B`, or `C<M>=A+B` with any mask and where `C` is initially empty.

- If both `A` and `B` are full (all entries present), then the rules for `eWiseMult` in Section 13.3 are used instead.
- If the operator is positional (`GxB_FIRSTI` and related) then `C` is not iso.

- If a and b differ (when typecasted to the type of `C`), then `C` is not iso.
- If $c = f(a, b) = a = b$ holds, then `C` is iso, where $f(a, b)$ is the operator.
- Otherwise, `C` is not iso.

13.5 Iso matrices from eWiseUnion

`GxB_eWiseUnion` is very similar to `GrB_eWiseAdd`, but the rules for when the result is iso-valued are very different.

- If both `A` and `B` are full (all entries present), then the rules for `eWiseMult` in Section 13.3 are used instead.
- If the operator is positional (`GxB_FIRSTI` and related) then `C` is not iso.
- If the op is `PAIR` (same as `ONEB`), then `C` is iso with $c = 1$.
- If `B` is iso and the op is `SECOND` or `ANY`, and the input scalar `beta` matches b (the iso-value of `B`), then `C` is iso with $c = b$.
- If `A` is iso and the op is `FIRST` or `ANY`, and the input scalar `alpha` matches a (the iso-value of `A`), then `C` is iso with $c = a$.
- If both `A` and `B` are iso, and $f(a, b) = f(\alpha, b) = f(a, \beta)$, then `C` is iso with $c = f(a, b)$.
- Otherwise, `C` is not iso.

13.6 Reducing iso matrices to a scalar or vector

If `A` is iso with e entries, reducing it to a scalar takes $O(\log(e))$ time, regardless of the monoid used to reduce the matrix to a scalar. Reducing `A` to a vector `c` is the same as the matrix-vector multiply `c=A*x` or `c=A'*x`, depending on the descriptor, where `x` is an iso full vector (refer to Section 13.2).

13.7 Iso matrices from apply

Let `C=f(A)` denote the application of a unary operator `f`, and let `C=f(A,s)` and `C=f(s,A)` denote the application of a binary operator with `s` a scalar.

- If the operator is positional (`GxB_POSITION*`, `GxB_FIRSTI`, and related) then `C` is not iso.
- If the operator is `ONE` or `PAIR` (same as `ONEB`), then `C` iso with $c = 1$.
- If the operator is `FIRST` or `ANY` with `C=f(s,A)`, then `C` iso with $c = s$.
- If the operator is `SECOND` or `ANY` with `C=f(A,s)`, then `C` iso with $c = s$.
- If `A` is iso then `C` is iso, with the following value of c:
  - If the op is `IDENTITY`, then $c = a$.

  - If the op is unary with `C=f(A)`, then $c = f(a)$.
  - If the op is binary with `C=f(s,A)`, then $c = f(s, a)$.
  - If the op is binary with `C=f(A,s)`, then $c = f(a, s)$.

- Otherwise, `C` is not iso.

13.8 Iso matrices from select

Let `C=select(A)` denote the application of a `GrB_IndexUnaryOp` operator in `GrB_select`.

- If `A` is iso, then `C` is iso with $c = a$.
- If the operator is any `GrB_VALUE*_BOOL` operator, with no typecasting, and the test is true only for a single boolean value, then `C` is iso.
- If the operator is `GrB_VALUEEQ_*`, with no typecasting, then `C` is iso, with $c = t$ where t is the value of the scalar `y`.
- If the operator is `GrB_VALUELE_UINT*`, with no typecasting, and the scalar `y` is zero, then `C` is iso with $c = 0$.
- Otherwise, `C` is not iso.

13.9 Iso matrices from assign and subassign

These rules are somewhat complex. Consider the assignment `C<M>(I,J)=...` with `GrB_assign`. Internally, this assignment is converted into `C(I,J)<M(I,J)>=...` and then `GxB_subassign` is used. Thus, all of the rules below assume the form `C(I,J)<M>=...` where `M` has the same size as the submatrix `C(I,J)`.

13.9.1 Assignment with no accumulator operator

If no accumulator operator is present, the following rules are used.

- For matrix assignment, `A` must be iso. For scalar assignment, the single scalar is implicitly expanded into an iso matrix `A` of the right size. If these rules do not hold, `C` is not iso.
- If `A` is not iso, or if `C` is not iso on input, then `C` is not iso on output.
- If `C` is iso or empty on input, and `A` is iso (or scalar assignment is begin performed) and the iso values c and a (or the scalar s) match, then the following forms of assignment result in an iso matrix `C` on output:
  - `C(I,J) = scalar`
  - `C(I,J)<M> = scalar`
  - `C(I,J)<!M> = scalar`

  - `C(I,J)<M,replace> = scalar`
  - `C(I,J)<!M,replace> = scalar`
  - `C(I,J) = A`
  - `C(I,J)<M> = A`
  - `C(I,J)<!M> = A`
  - `C(I,J)<M,replace> = A`
  - `C(I,J)<!M,replace> = A`

- For these forms of assignment, `C` is always iso on output, regardless of its iso property on input:
  - `C = scalar`
  - `C<M,struct>=scalar`; C empty on input.
  - `C<C,struct>=scalar`

- For these forms of assignment, `C` is always iso on output if `A` is iso:
  - `C = A`
  - `C<M,str> = A`; C empty on input.

13.9.2 Assignment with an accumulator operator

If an accumulator operator is present, the following rules are used. Positional operators (`GxB_FIRSTI` and related) cannot be used as accumulator operators, so these rules do not consider that case.

- For matrix assignment, `A` must be iso. For scalar assignment, the single scalar is implicitly expanded into an iso matrix `A` of the right size. If these rules do not hold, `C` is not iso.
- For these forms of assignment `C` is iso if `C` is empty on input, or if $c = c + a$ for the where a is the iso value of `A` or the value of the scalar for scalar assignment.
  - `C(I,J) += scalar`
  - `C(I,J)<M> += scalar`
  - `C(I,J)<!M> += scalar`
  - `C(I,J)<M,replace> += scalar`
  - `C(I,J)<!M,replace> += scalar`
  - `C(I,J)<M,replace> += A`
  - `C(I,J)<!M,replace> += A`
  - `C(I,J) += A`
  - `C(I,J)<M> += A`
  - `C(I,J)<!M> += A`
  - `C += A`

13.10 Iso matrices from build methods

`GxB_Matrix_build_Scalar` and `GxB_Vector_build_Scalar` always construct an iso matrix/vector.

`GrB_Matrix_build` and `GrB_Vector_build` can also construct iso matrices and vectors. A non-iso matrix/vector is constructed first, and then the entries are checked to see if they are all equal. The resulting iso-valued matrix/vector will be efficient to use and will use less memory than a non-iso matrix/vector. However, constructing an iso matrix/vector with `GrB_Matrix_build` and `GrB_Vector_build` will take more time and memory than constructing the matrix/vector with `GxB_Matrix_build_Scalar` or `GxB_Vector_build_Scalar`.

13.11 Iso matrices from other methods

- For `GrB_Matrix_dup` and `GrB_Vector_dup`, the output matrix/vector has the same iso property as the input matrix/vector.
- `GrB_*_setElement_*` preserves the iso property of the matrix/vector it modifies, if the input scalar is equal to the iso value of the matrix/vector. If the matrix or vector has no entries, the first call to `setElement` makes it iso. This allows a sequence of `setElement` calls with the same scalar value to create an entire iso matrix or vector, if starting from an empty matrix or vector.
- `GxB_Matrix_concat` constructs an iso matrix as its result if all input tiles are either empty or iso.
- `GxB_Matrix_split` constructs its output tiles as iso if its input matrix is iso.
- `GxB_Matrix_diag` and `GrB_Matrix_diag` construct an iso matrix if its input vector is iso.
- `GxB_Vector_diag` constructs an iso vector if its input matrix is iso.
- `GrB_*extract` constructs an iso matrix/vector if its input matrix/vector is iso.
- `GrB_transpose` constructs an iso matrix if its input is iso.
- The `GxB_import/export/pack/unpack` methods preserve the iso property of their matrices/vectors.

13.12 Iso matrices not exploited

There are many cases where an matrix may have the iso property but it is not detected by SuiteSparse:GraphBLAS. For example, if `A` is non-iso, `C=A(I,J)` from `GrB_extract` may be iso, if all entries in the extracted submatrix have the same value. Future versions of SuiteSparse:GraphBLAS may extend the rules described in this section to detect these cases.

14 Performance

Getting the best performance out of an algorithm that relies on GraphBLAS can depend on many factors. This section describes some of the possible performance pitfalls you can hit when using SuiteSparse:GraphBLAS, and how to avoid them (or at least know when you've encountered them).

14.1 The burble is your friend

Turn on the burble with `GxB_set (GxB_BURBLE, true)`. You will get a single line of output from each (significant) call to GraphBLAS. The burble output can help you detect when you are likely using sub-optimal methods, as described in the next sections.

14.2 Data types and typecasting

Avoid mixing data types and relying on typecasting as much as possible. SuiteSparse:GraphBLAS has a set of highly-tuned kernels for each data type, and many operators and semirings, but there are too many combinations to generate ahead of time. If typecasting is required, or if SuiteSparse:GraphBLAS does not have a kernel for the specific operator or semiring, the word `generic` will appear in the burble. The generic methods rely on function pointers for each operation on every scalar, so they are slow. A future JIT will avoid this problem.

The only time that typecasting is fast is when computing `C=A` via `GrB_assign` or `GrB_apply`, where the data types of `C` and `A` can differ. In this case, one of $13^2 = 169$ kernels are called, each of which performs the specific typecasting requested, without relying on function pointers.

14.3 Matrix data structures: sparse, hypersparse, bitmap, or full

SuiteSparse:GraphBLAS tries to automatically determine the best data structure for your matrices and vectors, selecting between sparse, hypersparse, bitmap, and full formats. By default, all 4 formats can be used. A matrix typically starts out hypersparse when it is created by `GrB_Matrix_new`, and then changes during its lifetime, possibly taking on all four different formats at different times. This can be modified via `GxB_set`. For example, this line of code:

```
GxB_set (A, GxB_SPARSITY_CONTROL, GxB_SPARSE + GxB_BITMAP) ;
```

tells SuiteSparse that the matrix `A` can be held in either sparse or bitmap format (at its discretion), but not hypersparse or full. The bitmap format will be used if the matrix has enough entries, or sparse otherwise. Sometimes this selection is best controlled by the user algorithm, so a single format can be requested:

```
GxB_set (A, GxB_SPARSITY_CONTROL, GxB_SPARSE) ;
```

This ensures that SuiteSparse will primarily use the sparse format. This is still just a hint, however. The data structure is opaque and SuiteSparse is free to choose otherwise. In particular, if you insist on using only the `GxB_FULL` format, then that format is used when all entries are present. However, if the matrix is not actually full with all entries present, then the bitmap format is used instead. The full format does not preserve the sparsity structure in this case. Any GraphBLAS library must preserve the proper structure, per the C Specification. This is critical in a graph algorithm, since an edge (i, j) of weight zero, say, is not the same as no edge (i, j) at all.

14.4 Matrix formats: by row or by column, or using the transpose of a matrix

By default, SuiteSparse uses a simple rule: all matrices are held by row, unless the consist of a single column, in which case they are held by column. All vectors are treated as if they are n-by-1 matrices with a single column. Changing formats from row-oriented to column-oriented can have significant performance implications, so SuiteSparse never tries to outguess the application. It just uses this simple rule.

However, there are cases where changing the format can greatly improve performance. There are two ways to handle this, which in the end are equivalent in the SuiteSparse internals. You can change the format (row to column oriented, or visa versa), or work with the explicit transpose of a matrix in the same storage orientation.

There are cases where SuiteSparse must explicitly transpose an input matrix, or the output matrix, in order to perform a computation. For example, if all matrices are held in row-oriented fashion, SuiteSparse does not have a method for computing `C=A'*B`, where `A` is transposed. Thus, SuiteSparse either computes a temporary transpose of its input matrix `AT=A` and then `C=AT*B`, or it swaps the computations, performing `C=(B'*A)'`, which requires an explicit transpose of `BT=B`, and a transpose of the final result to obtain `C`.

These temporary transposes are costly to compute, taking time and memory. They are not kept, but are discarded when the method returns to the user application. If you see the term `transpose` in the burble output, and if you need to perform this computation many times, try constructing your own explicit transpose, say `AT=A'`, via `GrB_transpose`, or create a copy of `A` but held in another orientation via `GxB_set`. For example, assuming the default matrix format is by-row, and that `A` is `m`-by-`n` of type `GrB_FP32`:

```
// method 1: AT = A'
GrB_Matrix_new (AT, GrB_FP32, n, m) ;
GrB_transpose (AT, NULL, NULL, A, NULL) ;

// method 2: A2 = A but held by column instead of by row
// note: doing the set before the assign is faster than the reverse
GrB_Matrix_new (A2, GrB_FP32, m, n) ;
GxB_set (A2, GxB_FORMAT, GxB_BY_COL) ;
GrB_assign (A2, NULL, NULL, A, GrB_ALL, m, GrB_ALL, n, NULL) ;
```

Internally, the data structure for `AT` and `A2` are nearly identical (that is, the tranpose of `A` held in row format is the same as `A` held in column format). Using either of them in subsequent calls to GraphBLAS will allow SuiteSparse to avoid computing an explicit

transpose. The two matrices `AT` and `A2` do differ in one very significant way: their dimensions are different, and they behave differement mathematically. Computing `C=A'*B` using these matrices would differ:

```
// method 1: C=A'*B using AT
GrB_mxm (C, NULL, NULL, semiring, AT, B, NULL) ;

// method 2: C=A'*B using A2
GrB_mxm (C, NULL, NULL, semiring, A2, B, GrB_DESC_T0) ;
```

The first method computes `C=AT*B`. The second method computes `C=A2'*B`, but the result of both computations is the same, and internally the same kernels will be used.

14.5 Push/pull optimization

Closely related to the discussion above on when to use a matrix or its transpose is the exploitation of "push/pull" direction optimization. In linear algebraic terms, this is simply deciding whether to multiply by the matrix or its transpose. Examples can be see in the BFS and Betweeness-Centrality methods of LAGraph. Here is the BFS kernel:

```
int sparsity = do_push ? GxB_SPARSE : GxB_BITMAP ;
GxB_set (q, GxB_SPARSITY_CONTROL, sparsity) ;
if (do_push)
{
    // q'{!pi} = q'*A
    GrB_vxm (q, pi, NULL, semiring, q, A, GrB_DESC_RSC) ;
}
else
{
    // q{!pi} = AT*q
    GrB_mxv (q, pi, NULL, semiring, AT, q, GrB_DESC_RSC) ;
}
```

The call to `GxB_set` is optional, since SuiteSparse will likely already determine that a bitmap format will work best when the frontier `q` has many entries, which is also when the pull step is fastest. The push step relies on a sparse vector times sparse matrix method originally due to Gustavson. The output is computed as a set union of all rows `A(i,:)` where `q(i)` is present on input. This set union is very fast when `q` is very sparse. The pull step relies on a sequence of dot product computations, one per possible entry in the output `q`, and it uses the matrix `AT` which is a row-oriented copy of the explicit transpose of the adjacency matrix `A`.

Mathematically, the results of the two methods are identical, but internally, the data format of the input matrices is very different (using `A` held by row, or `AT` held by row which is the same as a copy of `A` that is held by column), and the algorithms used are very different.

14.6 Computing with full matrices and vectors

Sometimes the best approach to getting the highest performance is to use dense vectors, and occassionaly dense matrices are tall-and-thin or short-and-fat. Packages such as Julia, Octave, or MATLAB, when dealing with the conventional plus-times semirings, assume that multiplying a sparse matrix `A` times a dense vector `x`, `y=A*x`, will result in a dense vector `y`. This is not always the case, however. GraphBLAS must always return a result that respects the sparsity structure of the output matrix or vector. If the ith row of `A` has no entries then `y(i)` must not appear as an entry in the vector `y`, so it cannot be held as a full vector. As a result, the following computation can be slower than it could be:

```
GrB_mxv (y, NULL, NULL, semiring, A, x, NULL) ;
```

SuiteSparse must do extra work to compute the sparsity of this vector `y`, but if this is not needed, and `y` can be padded with zeros (or the identity value of the monoid, to be precise), a faster method can be used, by relying on the accumulator. Instead of computing `y=A*x`, set all entries of `y` to zero first, and then compute `y+=A*x` where the accumulator operator and type matches the monoid of the semiring. SuiteSparse has special kernels for this case; you can see them in the burble as `F+=S*F` for example.

```
// y = 0
GrB_assign (y, NULL, NULL, 0, GrB_ALL, n, NULL) ;
// y += A*x
GrB_mxv (y, NULL, GrB_PLUS_FP32, GrB_PLUS_TIMES_SEMIRING_FP32, A, x, NULL) ;
```

You can see this computation in the LAGraph PageRank method, where all entries of `r` are set to the `teleport` scalar first.

```
for (iters = 0 ; iters < itermax && rdiff > tol ; iters++)
{
    // swap t and r ; now t is the old score
    GrB_Vector temp = t ; t = r ; r = temp ;
    // w = t ./ d
    GrB_eWiseMult (w, NULL, NULL, GrB_DIV_FP32, t, d, NULL) ;
    // r = teleport
    GrB_assign (r, NULL, NULL, teleport, GrB_ALL, n, NULL) ;
    // r += A'*w
    GrB_mxv (r, NULL, GrB_PLUS_FP32, LAGraph_plus_second_fp32, AT, w, NULL) ;
    // t -= r
    GrB_assign (t, NULL, GrB_MINUS_FP32, r, GrB_ALL, n, NULL) ;
    // t = abs (t)
    GrB_apply (t, NULL, NULL, GrB_ABS_FP32, t, NULL) ;
    // rdiff = sum (t)
    GrB_reduce (&rdiff, NULL, GrB_PLUS_MONOID_FP32, t, NULL) ;
}
```

SuiteSparse exploits the iso-valued property of the scalar-to-vector assignment of `y=0`, or `r=teleport`, and performs these assignments in O(1) time and space. Because the `r` vector start out as full on input to `GrB_mxv`, and because there is an accumulatr with no mask, no entries in the input/output vector `r` will be deleted, even if `A` has empty rows. The call to `GrB_mxv` exploits this, and is able to use a fast kernel for this computation. SuiteSparse does not need to compute the sparsity pattern of the vector `r`.

14.7 Iso-valued matrices and vectors

Using iso-valued matrices and vectors is always faster than using matrices and vectors whose entries can have different values. Iso-valued matrices are very important in graph algorithms. For example, an unweighted graph is best represented as an iso-valued sparse matrix, and unweighted graphs are very common. The burble output, or the `GxB_print`, `GxB_Matrix_iso`, or `GxB_Vector_iso` can all be used to report whether or not your matrix or vector is iso-valued.

Sometimes a matrix or vector may have values that are all the same, but SuiteSparse hasn't detected this. If this occurs, you can force a matrix or vector to be iso-valued by assigning a single scalar to all its entries.

```
// C<s(C)> = 3.14159
GrB_assign (C, C, NULL, 3.14159, GrB_ALL, m, GrB_ALL, n, GrB_DESC_S) ;
```

The matrix `C` is used as its own mask. The descriptor is essential here, telling the mask to be used in a structural sense, without regard to the values of the entries in the mask. This assignment sets all entries that already exist in `C` to be equal to a single value, 3.14159. The sparsity structure of `C` does not change. Of course, any scalar can be used; the value 1 is common for unweighted graphs. SuiteSparse:GraphBLAS performs the above assignment in O(1) time and space, independent of the dimension of `C` or the number of entries in contains.

14.8 User-defined types and operators

These are currently slow. Once SuiteSparse:GraphBLAS employs a JIT accelerator, these data types and operators will be just as fast as built-in types and operators. This work is in progress for the GPU, in CUDA, in collaboration with Joe Eaton and Corey Nolet.

15 Examples

NOTE: The programs in the `Demo` folder are not always the fastest methods. They are simple methods for illustration only, not performance. Do not benchmark them. Refer to the latest (draft) `LAGraph` package for the fastest methods. Be sure to use the right combination of package versions between LAGraph and SuiteSparse:GraphBLAS. Contact the author (davis@tamu.edu) if you have any questions about how to properly benchmark LAGraph + SuiteSparse:GraphBLAS.

Several examples of how to use GraphBLAS are listed below. They all appear in the `Demo` folder of SuiteSparse:GraphBLAS.

1. creating a random matrix
2. creating a finite-element matrix
3. reading a matrix from a file
4. complex numbers as a user-defined type
5. matrix import/export

Additional examples appear in the newly created LAGraph project, currently in progress.

15.1 LAGraph

The LAGraph project is a community-wide effort to create graph algorithms based on GraphBLAS (any implementation of the API, not just SuiteSparse: GraphBLAS). Some of the algorithms and utilities in LAGraph are listed in the table below. Many additional algorithms are planned. Refer to https://github.com/GraphBLAS/LAGraph for a current list of algorithms. All functions in the `Demo/` folder in SuiteSparse:GraphBLAS will eventually be translated into algorithms or utilities for LAGraph, and then removed from `GraphBLAS/Demo`.

To use LAGraph with SuiteSparse:GraphBLAS, place the two folders `LAGraph` and `GraphBLAS` in the same parent directory. This allows the `cmake` script in LAGraph to find the copy of GraphBLAS. Alternatively, the GraphBLAS source could be placed anywhere, as long as `sudo make install` is performed.

15.2 Creating a random matrix

The `random_matrix` function in the `Demo` folder generates a random matrix with a specified dimension and number of entries, either symmetric or unsymmetric, and with or without self-edges (diagonal entries in the matrix). It relies on `simple_rand*` functions in the `Demo` folder to provide a portable random number generator that creates the same sequence on any computer and operating system.

`random_matrix` can use one of two methods: `GrB_Matrix_setElement` and `GrB_Matrix_build`. The former method is very simple to use:

```
GrB_Matrix_new (&A, GrB_FP64, nrows, ncols) ;
for (int64_t k = 0 ; k < ntuples ; k++)
{
    GrB_Index i = simple_rand_i ( ) % nrows ;
    GrB_Index j = simple_rand_i ( ) % ncols ;
    if (no_self_edges && (i == j)) continue ;
    double x = simple_rand_x ( ) ;
    // A (i,j) = x
    GrB_Matrix_setElement (A, x, i, j) ;
    if (make_symmetric)
    {
        // A (j,i) = x
        GrB_Matrix_setElement (A, x, j, i) ;
    }
}
```

The above code can generate a million-by-million sparse `double` matrix with 200 million entries in 66 seconds (6 seconds of which is the time to generate the random `i`, `j`, and `x`), including the time to finish all pending computations. The user application does not need to create a list of all the tuples, nor does it need to know how many entries will appear in the matrix. It just starts from an empty matrix and adds them one at a time in arbitrary order. GraphBLAS handles the rest. This method is not feasible in MATLAB.

The next method uses `GrB_Matrix_build`. It is more complex to use than `setElement` since it requires the user application to allocate and fill the tuple lists, and it requires knowledge of how many entries will appear in the matrix, or at least a good upper bound, before the matrix is constructed. It is slightly faster, creating the same matrix in 60 seconds, 51 seconds of which is spent in `GrB_Matrix_build`.

```
GrB_Index *I, *J ;
double *X ;
int64_t s = ((make_symmetric) ? 2 : 1) * nedges + 1 ;
I = malloc (s * sizeof (GrB_Index)) ;
J = malloc (s * sizeof (GrB_Index)) ;
X = malloc (s * sizeof (double   )) ;
if (I == NULL || J == NULL || X == NULL)
{
    // out of memory
    if (I != NULL) free (I) ;
    if (J != NULL) free (J) ;
    if (X != NULL) free (X) ;
    return (GrB_OUT_OF_MEMORY) ;
}
int64_t ntuples = 0 ;
for (int64_t k = 0 ; k < nedges ; k++)
{
    GrB_Index i = simple_rand_i ( ) % nrows ;
```

```
        GrB_Index j = simple_rand_i ( ) % ncols ;
        if (no_self_edges && (i == j)) continue ;
        double x = simple_rand_x ( ) ;
        // A (i,j) = x
        I [ntuples] = i ;
        J [ntuples] = j ;
        X [ntuples] = x ;
        ntuples++ ;
        if (make_symmetric)
        {
            // A (j,i) = x
            I [ntuples] = j ;
            J [ntuples] = i ;
            X [ntuples] = x ;
            ntuples++ ;
        }
    }
    GrB_Matrix_build (A, I, J, X, ntuples, GrB_SECOND_FP64) ;
```

The equivalent `sprandsym` function in MATLAB takes 150 seconds, but `sprandsym` uses a much higher-quality random number generator to create the tuples `[I,J,X]`. Considering just the time for `sparse(I,J,X,n,n)` in `sprandsym` (equivalent to `GrB_Matrix_build`), the time is 70 seconds. That is, each of these three methods, `setElement` and `build` in SuiteSparse:GraphBLAS, and `sparse` in MATLAB, are equally fast.

15.3 Creating a finite-element matrix

Suppose a finite-element matrix is being constructed, with `k=40,000` finite-element matrices, each of size `8`-by-`8`. The following operations (in pseudo-MATLAB notation) are very efficient in SuiteSparse:GraphBLAS.

```
A = sparse (m,n) ; % create an empty n-by-n sparse GraphBLAS matrix
for i = 1:k
    construct a 8-by-8 sparse or dense finite-element F
    I and J define where the matrix F is to be added:
    I = a list of 8 row indices
    J = a list of 8 column indices
    % using GrB_assign, with the 'plus' accum operator:
    A (I,J) = A (I,J) + F
end
```

If this were done in MATLAB or in GraphBLAS with blocking mode enabled, the computations would be extremely slow. A far better approach is to construct a list of tuples `[I,J,X]` and to use `sparse(I,J,X,n,n)`. This is identical to creating the same list of tuples in GraphBLAS and using the `GrB_Matrix_build`, which is equally fast.

In SuiteSparse:GraphBLAS, the performance of both methods is essentially identical, and roughly as fast as `sparse` in MATLAB. Inside SuiteSparse:GraphBLAS, `GrB_assign` is doing the same thing. When performing `A(I,J)=A(I,J)+F`, if it finds that it cannot quickly insert an update into the `A` matrix, it creates a list of pending tuples to be assembled later on. When the matrix is ready for use in a subsequent GraphBLAS operation (one that normally cannot use a matrix with pending computations), the tuples are assembled all at once via `GrB_Matrix_build`.

GraphBLAS operations on other matrices have no effect on when the pending updates of a matrix are completed. Thus, any GraphBLAS method or operation can be used to construct the `F` matrix in the example above, without affecting when the pending updates to `A` are completed.

The MATLAB `wathen.m` script is part of Higham's `gallery` of matrices [Hig02]. It creates a finite-element matrix with random coefficients for a 2D mesh of size `nx`-by-`ny`, a matrix formulation by Wathen [Wat87]. The pattern of the matrix is fixed; just the values are randomized. The GraphBLAS equivalent can use either `GrB_Matrix_build`, or `GrB_assign`. Both methods have good performance. The `GrB_Matrix_build` version below is about 15% to 20% faster than the MATLAB `wathen.m` function, regardless of the problem size. It uses the identical algorithm as `wathen.m`.

```
int64_t ntriplets = nx*ny*64 ;
I = malloc (ntriplets * sizeof (int64_t)) ;
J = malloc (ntriplets * sizeof (int64_t)) ;
X = malloc (ntriplets * sizeof (double )) ;
if (I == NULL || J == NULL || X == NULL)
{
    FREE_ALL ;
    return (GrB_OUT_OF_MEMORY) ;
}
ntriplets = 0 ;
for (int j = 1 ; j <= ny ; j++)
{
    for (int i = 1 ; i <= nx ; i++)
    {
        nn [0] = 3*j*nx + 2*i + 2*j + 1 ;
        nn [1] = nn [0] - 1 ;
        nn [2] = nn [1] - 1 ;
        nn [3] = (3*j-1)*nx + 2*j + i - 1 ;
        nn [4] = 3*(j-1)*nx + 2*i + 2*j - 3 ;
        nn [5] = nn [4] + 1 ;
        nn [6] = nn [5] + 1 ;
        nn [7] = nn [3] + 1 ;
        for (int krow = 0 ; krow < 8 ; krow++) nn [krow]-- ;
        for (int krow = 0 ; krow < 8 ; krow++)
        {
            for (int kcol = 0 ; kcol < 8 ; kcol++)
            {
```

```
                I [ntriplets] = nn [krow] ;
                J [ntriplets] = nn [kcol] ;
                X [ntriplets] = em (krow,kcol) ;
                ntriplets++ ;
            }
        }
    }
}
// A = sparse (I,J,X,n,n) ;
GrB_Matrix_build (A, I, J, X, ntriplets, GrB_PLUS_FP64) ;
```

The `GrB_assign` version has the advantage of not requiring the user application to construct the tuple list, and is almost as fast as using `GrB_Matrix_build`. The code is more elegant than either the MATLAB `wathen.m` function or its GraphBLAS equivalent above. Its performance is comparable with the other two methods, but slightly slower, being about 5% slower than the MATLAB `wathen`, and 20% slower than the GraphBLAS method above.

```
GrB_Matrix_new (&F, GrB_FP64, 8, 8) ;
for (int j = 1 ; j <= ny ; j++)
{
    for (int i = 1 ; i <= nx ; i++)
    {
        nn [0] = 3*j*nx + 2*i + 2*j + 1 ;
        nn [1] = nn [0] - 1 ;
        nn [2] = nn [1] - 1 ;
        nn [3] = (3*j-1)*nx + 2*j + i - 1 ;
        nn [4] = 3*(j-1)*nx + 2*i + 2*j - 3 ;
        nn [5] = nn [4] + 1 ;
        nn [6] = nn [5] + 1 ;
        nn [7] = nn [3] + 1 ;
        for (int krow = 0 ; krow < 8 ; krow++) nn [krow]-- ;
        for (int krow = 0 ; krow < 8 ; krow++)
        {
            for (int kcol = 0 ; kcol < 8 ; kcol++)
            {
                // F (krow,kcol) = em (krow, kcol)
                GrB_Matrix_setElement (F, em (krow,kcol), krow, kcol) ;
            }
        }
        // A (nn,nn) += F
        GrB_assign (A, NULL, GrB_PLUS_FP64, F, nn, 8, nn, 8, NULL) ;
    }
}
```

Since there is no Mask, and since GrB_REPLACE is not used, the call to GrB_assign in the example above is identical to GxB_subassign. Either one can be used, and their performance would be identical.

Refer to the wathen.c function in the Demo folder, which uses GraphBLAS to implement the two methods above, and two additional ones.

15.4 Reading a matrix from a file

See also LAGraph_mmread and LAGraph_mmwrite, which can read and write any matrix in Matrix Market format, and LAGraph_binread and LAGraph_binwrite, which read/write a matrix from a binary file. The binary file I/O functions are much faster than the read_matrix function described here, and also much faster than LAGraph_mmread and LAGraph_mmwrite.

The read_matrix function in the Demo reads in a triplet matrix from a file, one line per entry, and then uses GrB_Matrix_build to create the matrix. It creates a second copy with GrB_Matrix_setElement, just to test that method and compare the run times. Section 15.2 has already compared build versus setElement.

The function can return the matrix as-is, which may be rectangular or unsymmetric. If an input parameter is set to make the matrix symmetric, read_matrix computes A=(A+A')/2 if A is square (turning all directed edges into undirected ones). If A is rectangular, it creates a bipartite graph, which is the same as the augmented matrix, A = [0 A ; A' 0]. If C is an n-by-n matrix, then C=(C+C')/2 can be computed as follows in GraphBLAS, (the scale2 function divides an entry by 2):

```
GrB_Descriptor_new (&dt2) ;
GrB_Descriptor_set (dt2, GrB_INP1, GrB_TRAN) ;
GrB_Matrix_new (&A, GrB_FP64, n, n) ;
GrB_eWiseAdd (A, NULL, NULL, GrB_PLUS_FP64, C, C, dt2) ;    // A=C+C'
GrB_free (&C) ;
GrB_Matrix_new (&C, GrB_FP64, n, n) ;
GrB_UnaryOp_new (&scale2_op, scale2, GrB_FP64, GrB_FP64) ;
GrB_apply (C, NULL, NULL, scale2_op, A, NULL) ;             // C=A/2
GrB_free (&A) ;
GrB_free (&scale2_op) ;
```

This is of course not nearly as elegant as A=(A+A')/2 in MATLAB, but with minor changes it can work on any type and use any built-in operators instead of PLUS, or it can use any user-defined operators and types. The above code in SuiteSparse:GraphBLAS takes 0.60 seconds for the Freescale2 matrix, slightly slower than MATLAB (0.55 seconds).

Constructing the augmented system is more complicated using the GraphBLAS C API Specification since it does not yet have a simple way of specifying a range of row and column indices, as in A(10:20,30:50) in MATLAB (GxB_RANGE is a SuiteSparse:GraphBLAS extension that is not in the Specification). Using the C API in the Specification, the application must instead build a list of indices first, I=[10, 11 ... 20].

Thus, to compute the MATLAB equivalent of A = [0 A ; A' 0], index lists I and J must first be constructed:

```
int64_t n = nrows + ncols ;
I = malloc (nrows * sizeof (int64_t)) ;
J = malloc (ncols * sizeof (int64_t)) ;
// I = 0:nrows-1
// J = nrows:n-1
if (I == NULL || J == NULL)
{
    if (I != NULL) free (I) ;
    if (J != NULL) free (J) ;
    return (GrB_OUT_OF_MEMORY) ;
}
for (int64_t k = 0 ; k < nrows ; k++) I [k] = k ;
for (int64_t k = 0 ; k < ncols ; k++) J [k] = k + nrows ;
```

Once the index lists are generated, however, the resulting GraphBLAS operations are fairly straightforward, computing `A=[0 C ; C' 0]`.

```
GrB_Descriptor_new (&dt1) ;
GrB_Descriptor_set (dt1, GrB_INP0, GrB_TRAN) ;
GrB_Matrix_new (&A, GrB_FP64, n, n) ;
// A (nrows:n-1, 0:nrows-1) = C'
GrB_assign (A, NULL, NULL, C, J, ncols, I, nrows, dt1) ;
// A (0:nrows-1, nrows:n-1) = C
GrB_assign (A, NULL, NULL, C, I, nrows, J, ncols, NULL) ;
```

This takes 1.38 seconds for the `Freescale2` matrix, almost as fast as `A=[sparse(m,m) C ; C' sparse(n,n)]` in MATLAB (1.25 seconds).

Both calls to `GrB_assign` use no accumulator, so the second one causes the partial matrix `A=[0 0 ; C' 0]` to be built first, followed by the final build of `A=[0 C ; C' 0]`. A better method, but not an obvious one, is to use the `GrB_FIRST_FP64` accumulator for both assignments. An accumulator enables SuiteSparse:GraphBLAS to determine that that entries created by the first assignment cannot be deleted by the second, and thus it need not force completion of the pending updates prior to the second assignment.

SuiteSparse:GraphBLAS also adds a `GxB_RANGE` mechanism that mimics the MATLAB colon notation. This speeds up the method and simplifies the code the user needs to write to compute `A=[0 C ; C' 0]`:

```
int64_t n = nrows + ncols ;
GrB_Matrix_new (&A, xtype, n, n) ;
GrB_Index I_range [3], J_range [3] ;
I_range [GxB_BEGIN] = 0 ;
I_range [GxB_END  ] = nrows-1 ;
J_range [GxB_BEGIN] = nrows ;
J_range [GxB_END  ] = ncols+nrows-1 ;
// A (nrows:n-1, 0:nrows-1) += C'
GrB_assign (A, NULL, GrB_FIRST_FP64, // or NULL,
```

```
    C, J_range, GxB_RANGE, I_range, GxB_RANGE, dt1) ;
// A (0:nrows-1, nrows:n-1) += C
GrB_assign (A, NULL, GrB_FIRST_FP64, // or NULL,
    C, I_range, GxB_RANGE, J_range, GxB_RANGE, NULL) ;
```

Any operator will suffice because it is not actually applied. An operator is only applied to the set intersection, and the two assignments do not overlap. If an `accum` operator is used, only the final matrix is built, and the time in GraphBLAS drops slightly to 1.25 seconds. This is a very small improvement because in this particular case, SuiteSparse:GraphBLAS is able to detect that no sorting is required for the first build, and the second one is a simple concatenation. In general, however, allowing GraphBLAS to postpone pending updates can lead to significant reductions in run time.

15.5 User-defined types and operators

The `Demo` folder contains two working examples of user-defined types, first discussed in Section 6.1.1: `double complex`, and a user-defined `typedef` called `wildtype` with a `struct` containing a string and a 4-by-4 `float` matrix.

Double Complex: Prior to v3.3, GraphBLAS did not have a native complex type. It now appears as the `GxB_FC64` predefined type, but a complex type can also easily added as a user-defined type. The `Complex_init` function in the `usercomplex.c` file in the `Demo` folder creates the `Complex` type based on the ANSI C11 `double complex` type. It creates a full suite of operators that correspond to every built-in GraphBLAS operator, both binary and unary. In addition, it creates the operators listed in the following table, where D is `double` and C is `Complex`.

name	types	Octave/MATLAB equivalent	description
`Complex_complex`	$D \times D \to C$	`z=complex(x,y)`	complex from real and imag.
`Complex_conj`	$C \to C$	`z=conj(x)`	complex conjugate
`Complex_real`	$C \to D$	`z=real(x)`	real part
`Complex_imag`	$C \to D$	`z=imag(x)`	imaginary part
`Complex_angle`	$C \to D$	`z=angle(x)`	phase angle
`Complex_complex_real`	$D \to C$	`z=complex(x,0)`	real to complex real
`Complex_complex_imag`	$D \to C$	`z=complex(0,x)`	real to complex imag.

The `Complex_init` function creates two monoids (`Complex_add_monoid` and `Complex_times_monoid`) and a semiring `Complex_plus_times` that corresponds to the conventional linear algebra for complex matrices. The include file `usercomplex.h` in the `Demo` folder is available so that this user-defined `Complex` type can easily be imported into any other user application. When the user application is done, the `Complex_finalize` function frees the `Complex` type and its operators, monoids, and semiring. NOTE: the `Complex` type is not supported in this Demo in Microsoft Visual Studio.

Struct-based: In addition, the `wildtype.c` program creates a user-defined `typedef` of a `struct` containing a dense 4-by-4 `float` matrix, and a 64-character string. It constructs an additive monoid that adds two 4-by-4 dense matrices, and a multiplier operator

that multiplies two 4-by-4 matrices. Each of these 4-by-4 matrices is treated by GraphBLAS as a "scalar" value, and they can be manipulated in the same way any other GraphBLAS type can be manipulated. The purpose of this type is illustrate the endless possibilities of user-defined types and their use in GraphBLAS.

15.6 User applications using OpenMP or other threading models

An example demo program (`openmp_demo`) is included that illustrates how a multi-threaded user application can use GraphBLAS.

The results from the `openmp_demo` program may appear out of order. This is by design, simply to show that the user application is running in parallel. The output of each thread should be the same. In particular, each thread generates an intentional error, and later on prints it with `GrB_error`. It will print its own error, not an error from another thread. When all the threads finish, the leader thread prints out each matrix generated by each thread.

GraphBLAS can also be combined with user applications that rely on MPI, the Intel TBB threading library, POSIX pthreads, Microsoft Windows threads, or any other threading library. In all cases, GraphBLAS will be thread safe.

16 Compiling and Installing SuiteSparse:GraphBLAS

16.1 On Linux and Mac

GraphBLAS makes extensive use of features in the ANSI C11 standard, and thus a C compiler supporting this version of the C standard is required to use all features of GraphBLAS.

Any version of the Intel `icx` compiler is highly recommended. In most cases, the Intel `icx` and the Intel OpenMP library (`libiomp`) result in the best performance. The `gcc` and the GNU OpenMP library (`libgomp`) generally gives good performance: typically on par with icx but in a few special cases significantly slower. The Intel `icc` compiler is not recommended; it results in poor performance for `#pragma omp atomic`.

On the Mac (OS X), `clang` 8.0.0 in `Xcode` version 8.2.1 is sufficient, although earlier versions of `Xcode` may work as well. For the GNU `gcc` compiler, version 4.9 or later is required, but best performance is obtained in 9.3 or later. Version 3.13 or later of `cmake` is required; version 3.17 is preferred.

If you are using a pre-C11 ANSI C compiler, such as Microsoft Visual Studio, then the `_Generic` keyword is not available. SuiteSparse:GraphBLAS will still compile, but you will not have access to polymorphic functions such as `GrB_assign`. You will need to use the non-polymorphic functions instead.

To compile SuiteSparse:GraphBLAS, simply type `make` in the main GraphBLAS folder, which compiles the library with your default system compiler. This compile GraphBLAS using 8 threads, which will take a long time. To compile with more threads (40, for this example), use:

```
make JOBS=40
```

To use a non-default compiler with 4 threads:

```
make CC=icx CXX=icpx JOBS=4
```

GraphBLAS v6.1.3 and later use the `cpu_features` package by Google to determine if the target architecture supports AVX2 and/or AVX512F (on Intel x86_64 architectures only). In case you have build issues with this package, you can compile without it (and then AVX2 and AVX512F acceleration will not be used):

```
make CMAKE_OPTIONS='-DGBNCPUFEAT=1'
```

Without `cpu_features`, it is still possible to enable AVX2 and AVX512F. Rather than relying on run-time tests, you can use these flags to enable both AVX2 and AVX512F, without relying on `cpu_features`:

```
make CMAKE_OPTIONS='-DGBNCPUFEAT=1 -DGBAVX2=1 -DGBAVX512F=1'
```

To use multiple options, separate them by a space. For example, to build just the library but not `cpu_features`, and to enable AVX2 but not AVX512F, and use 40 threads to compile:

```
make CMAKE_OPTIONS='-DGBNCPUFEAT=1 -DGBAVX2=1' JOBS=40
```

After compiling the library, you can compile the demos with `make all` and then `make run`.

If `cmake` or `make` fail, it might be that your default compiler does not support ANSI C11. Try another compiler. For example, try one of these options. Go into the `build` directory and type one of these:

```
CC=gcc cmake ..
CC=gcc-11 cmake ..
CC=xlc cmake ..
CC=icx cmake ..
```

You can also do the following in the top-level GraphBLAS folder instead:

```
CC=gcc make
CC=gcc-11 make
CC=xlc make
CC=icx make
```

For faster compilation, you can specify a parallel make. For example, to use 32 parallel jobs and the `gcc` compiler, do the following:

```
JOBS=32 CC=gcc make
```

If you do not have `cmake`, refer to Section 16.8.

16.2 More details on the Mac

SuiteSparse:GraphBLAS requires OpenMP for its internal parallelism, but OpenMP is not on the Mac by default.

If you have the Intel compiler and OpenMP library, then use the following in the top-level `GraphBLAS` folder. OpenMP will be found automatically:

```
make CC=icc CXX=icpc
```

The following instructions work on MacOS Big Sur (v11.3) and MacOS Monterey (12.1), using cmake 3.13 or later:

First install Xcode (see https://developer.apple.com/xcode, and then install the command line tools for Xcode:

```
cd /Applications/Utilities
xcode-select |install
```

Next, install brew, at https://brew.sh.

If not used for the MATLAB mexFunction interface, a recent update of the Apple Clang compiler now works with `libomp` and the `GraphBLAS/CMakeLists.txt`. To use the MATLAB mexFunction, however, you must use `gcc` (`gcc-11` is recommended). Using Clang will result in a segfault when you attempt to use the `@GrB` interface in MATLAB.

With MacOS Big Sur install `gcc-11`, `cmake`, and OpenMP, and then compile GraphBLAS. cmake 3.13 or later is required. For the MATLAB mexFunctions, you must use `gcc-11`; the `libomp` from `brew` will allow you to compile the mexFunctions but they will not work properly.

```
brew install cmake
brew install libomp
brew install gcc
cd GraphBLAS/GraphBLAS
make CC=gcc-11 CXX=g++-11 JOBS=8
```

The above instructions assume MATLAB R2021a, using `libgraphblas_renamed.dylib`, since that version of MATLAB includes its own copy of SuiteSparse:GraphBLAS (`libmwgraphblas.dylib`) but at version v3.3.3, not the latest version.

Next, compile the MATLAB mexFunctions. I had to edit this file first:

```
/Users/davis/Library/Application Support/MathWorks/MATLAB/R2021a/mex_C_maci64.xml
```

where you would replace `davis` with your MacOS user name. Change lines 4 and 18, where both cases of `MACOSX_DEPLOYMENT_TARGET=10.14` must become `MACOSX_DEPLOYMENT_TARGET=11.3`. Otherwise, MATLAB complains that the `libgraphblas_renamed.dylib` was built for 11.3 but linked for 10.14.

Next, type the following in the MATLAB Command Window:

```
cd GraphBLAS/GraphBLAS/@GrB/private
gbmake
```

Then add the paths to your `startup.m` file (usually in `~/Documents/MATLAB/startup.m`). For example, my path is:

```
addpath ('/Users/davis/GraphBLAS/GraphBLAS') ;
addpath ('/Users/davis/GraphBLAS/GraphBLAS/build') ;
```

Finally, you can run the tests to see if your installation works:

```
cd ../../test
gbtest
```

16.3 On the ARM64 architecture

You may encounter a compiler error on the ARM64 architecture when using the `gcc` compiler, versions 6.x and earlier. This error was encountered on ARM64 Linux with gcc 6.x:

```
'In function GrB_Matrix_apply_BinaryOp1st_Scalar.part.1':
GrB_Matrix_apply.c:(.text+0x210): relocation truncated to
fit: R_AARCH64_CALL26 against '.text.unlikely'
```

For the ARM64, this error is silenced with gcc v7.x and later, at least on Linux.

16.4 On Microsoft Windows

SuiteSparse:GraphBLAS is now ported to Microsoft Visual Studio. However, that compiler is not ANSI C11 compliant. As a result, GraphBLAS on Windows will have a few minor limitations.

- The MS Visual Studio compiler does not support the `_Generic` keyword, required for the polymorphic GraphBLAS functions. So for example, you will need to use `GrB_Matrix_free` instead of just `GrB_free`.
- Variable-length arrays are not supported, so user-defined types are limited to 128 bytes in size. This can be changed by editing `GB_VLA_MAXSIZE` in `Source/GB_compiler.h`, and recompiling SuiteSparse:GraphBLAS.
- AVX acceleration is not enabled.

If you use a recent `gcc` or `icx` compiler on Windows other than the Microsoft Compiler (`cl`), these limitations can be avoided.

The following instructions apply to Windows 10, CMake 3.16, and Visual Studio 2019, but may work for earlier versions.

1. Install CMake 3.16 or later, if not already installed. See https://cmake.org/ for details.
2. Install Microsoft Visual Studio, if not already installed. See https://visualstudio.microsoft.com/ for details. Version 2019 is preferred, but earlier versions may also work.
3. Open a terminal window and type this in the `SuiteSparse/GraphBLAS/build` folder:

   ```
   cmake ..
   ```

4. The `cmake` command generates many files in `SuiteSparse/GraphBLAS/build`, and the file `graphblas.sln` in particular. Open the generated `graphblas.sln` file in Visual Studio.

5. Optionally: right-click `graphblas` in the left panel (Solution Explorer) and select properties; then navigate to `Configuration Properties`, `C/C++`, `General` and change the parameter `Multiprocessor Compilation` to `Yes (/MP)`. Click `OK`. This will significantly speed up the compilation of GraphBLAS.
6. Select the `Build` menu item at the top of the window and select `Build Solution`. This should create a folder called `Release` and place the compiled `graphblas.dll`, `graphblas.lib`, and `graphblas.exp` files there. Please be patient; some files may take a while to compile and sometimes may appear to be stalled. Just wait.
7. Add the `GraphBLAS/build/Release` folder to the Windows System path:
    - Open the `Start Menu` and type `Control Panel`.
    - Select the `Control Panel` app.
    - When the app opens, select `System and Security`.
    - Under `System and Security`, select `System`.
    - From the top left side of the `System` window, select `Advanced System Settings`. You may have to authenticate at this step.
    - The `Systems Properties` window should appear with the `Advanced` tab selected; select `Environment Variables`.
    - The `Environment Variables` window displays 2 sections, one for `User` variables and the other for `System` variables. Under the `Systems` variable section, scroll to and select `Path`, then select `Edit`. A editor window appears allowing to add, modify, delete or re-order the parts of the `Path`.
    - Add the full path of the `GraphBLAS\build\Release` folder (typically starting with `C:\Users\you\...`, where `you` is your Windows username) to the `Path`.
    - If the above steps do not work, you can instead copy the `graphblas.*` files from `GraphBLAS\build\Release` into any existing folder listed in your `Path`.
8. The `GraphBLAS/Include/GraphBLAS.h` file must be included in user applications via `#include "GraphBLAS.h"`. This is already done for you in the Octave/MATLAB interface discussed in the next section.

16.5 Compiling the Octave/MATLAB interface (for Octave, and for MATLAB R2020a and earlier)

I'm working closely with John Eaton (the primary developer of Octave) to enable SuiteSparse:GraphBLAS to work with Octave, and thus Octave 7 is required. The latest version of Octave is 6.4.0, so you need to download and install the development version of Octave 7 to use SuiteSparse:GraphBLAS within Octave.

First, compile the SuiteSparse:GraphBLAS dynamic library (`libgraphblas.so` for Linux, `libgraphblas.dylib` for Mac, or `graphblas.dll` for Windows), as described in the prior two subsections.

On the Mac, SuiteSparse:GraphBLAS v6.1.4 and Octave 7 will work Apple Silicon (thanks to Gábor Szárnyas). Here are his instructions (replicated from https://github.com/DrTimothyAldenDavis/GraphBLAS/issues/90); do these in your Mac Terminal:

- Building Octave. Grab the brew formula:

```
wget https://raw.githubusercontent.com/Homebrew/homebrew-core/master/Formula/octave.rb
```

- Edit `octave.rb`.

  Add `"disable-docs"` to `args` (or ensure that you have a working texinfo installation). Edit Mercurial (`hg`) repository: switch from the `default` branch (containing code for Octave v8.0) to `stable` (v7.0). Then do:

```
brew install --head ./octave.rb
```

- Building the tests (`gbmake`). Grab the OpenMP binaries as described at https://mac.r-project.org/openmp/

```
curl -O https://mac.r-project.org/openmp/openmp-13.0.0-darwin21-Release.tar.gz
sudo tar fvxz openmp-13.0.0-darwin21-Release.tar.gz -C /
```

- Do the following to edit `gbmake.m`:

```
sed -i.bkp 's/-fopenmp/-Xclang -fopenmp/g' @GrB/private/gbmake.m
```

Once Octave 7 and SuiteSparse:GraphBLAS are compiled and installed, and `gbmake.m` is modified if needed for Octave 7 on the Mac, (or if using MATLAB) continue with the following instructions:

1. In the Octave/MATLAB command window:

```
cd GraphBLAS/GraphBLAS/@GrB/private
gbmake
```

2. Follow the remaining instructions in the `GraphBLAS/GraphBLAS/README.md` file, to revise your Octave/MATLAB path and `startup.m` file.
3. As a quick test, try the command `GrB(1)`, which creates and displays a 1-by-1 GraphBLAS matrix. For a longer test, do the following:

```
cd GraphBLAS/GraphBLAS/test
gbtest
```

4. In Windows, if the tests fail with an error stating that the mex file is invalid because the module could not be found, it means that MATLAB could not find the compiled `graphblas.lib`, `*.dll` or `*.exp` files in the `build/Release` folder. This can happen if your Windows System path is not set properly, or if Windows is not recognizing the `GraphBLAS/build/Release` folder (see Section 16.4) Or, you might not have permission to change your Windows System path. In this case, do the following in the MATLAB Command Window:

   ```
   cd GraphBLAS/build/Release
   GrB(1)
   ```

   After this step, the GraphBLAS library will be loaded into MATLAB. You may need to add the above lines in your `Documents/MATLAB/startup.m` file, so that they are done each time MATLAB starts. You will also need to do this after `clear all` or `clear mex`, since those MATLAB commands remove all loaded libraries from MATLAB.

   You might also get an error "the specified procedure cannot be found." This can occur if you have upgraded your GraphBLAS library from a prior version, and some of the compiled files `@GrB/private/*.mex*` are stale. Try the command `gbmake all` in the MATLAB Command Window, which forces all of the MATLAB interface to be recompiled. Or, try deleting all `@GrB/private/*.mex*` files and running `gbmake` again.

5. On Windows, the `casin`, `casinf`, `casinh`, and `casinhf` functions provided by Microsoft do not return the correct imaginary part. As a result, `GxB_ASIN_FC32`, `GxB_ASIN_FC64` `GxB_ASINH_FC32`, and `GxB_ASINH_FC64` do not work properly on Windows. This affects the `GrB/asin`, `GrB/acsc`, `GrB/asinh`, and `GrB/acsch`, functions in the MATLAB interface. See the MATLAB tests bypassed in `gbtest76.m` for details, in the
   `GraphBLAS/GraphBLAS/test` folder.

16.6 Compiling the Octave/MATLAB interface (for MATLAB R2021a and later)

MATLAB R2021a includes its own copy of SuiteSparse:GraphBLAS v3.3.3, as the file `libmwgraphblas.so`, which is used for the built-in `C=A*B` when both `A` and `B` are sparse (see the Release Notes of MATLAB R2021a, which discusses the performance gained in MATLAB by using GraphBLAS).

That's great news for the impact of GraphBLAS on MATLAB itself, and the domain of high performance computing in general, but it causes a linking problem when using this MATLAB interface for GraphBLAS. The two use different versions of the same library, and a segfault arises if the MATLAB interface for v4.x (or later) tries to link with the older GraphBLAS v3.3.3 library. Likewise, the built-in `C=A*B` causes a segfault if it tries to use the newer GraphBLAS v4.x (or later) libraries.

To resolve this issue, a second GraphBLAS library must be compiled, `libgraphblas_renamed`, where the internal symbols are all renamed so they do not conflict with the `libmwgraphblas` library. Then both libraries can co-exist in the same instance of MATLAB.

To do this, go to the `GraphBLAS/GraphBLAS` folder, containing the MATLAB interface. That folder contains a `CMakeLists.txt` file to compile the `libgraphblas_renamed` library. See the instructions for how to compile the C library `libgraphblas`, and repeat them but using the folder
`SuiteSparse/GraphBLAS/GraphBLAS/build` instead of
`SuiteSparse/GraphBLAS/build`.

This will compile the renamed SuiteSparse:GraphBLAS dynamic library (`libgraphblas_renamed.so` for Linux, `libgraphblas_renamed.dylib` for Mac, or `graphblas_renamed.dll` for Windows). These can be placed in the same system-wide location as the standard `libgraphblas` libraries, such as `/usr/local/lib` for Linux. The two pairs of libraries share the identical `GraphBLAS.h` include file.

Next, compile the MATLAB interface as described in Section 16.5. For any instructions in that Section that refer to the `GraphBLAS/build` folder (Linux and Mac) or `GraphBLAS/build/Release` (Windows), use
`GraphBLAS/GraphBLAS/build` (Linux and Mac) or
`GraphBLAS/GraphBLAS/build/Release` (Windows) instead.

The resulting functions for your `@GrB` object will now work just fine; no other changes are needed. You can even use the GraphBLAS mexFunctions compiled in MATLAB R2021a in earlier versions of MATLAB (such as R2020a).

16.7 Setting the C flags and using CMake

Next, do `make` in the `build` directory. If this still fails, see the `CMakeLists.txt` file. You can edit that file to pass compiler-specific options to your compiler. Locate this section in the `CMakeLists.txt` file. Use the `set` command in `cmake`, as in the example below, to set the compiler flags you need.

```
# check which compiler is being used.  If you need to make
# compiler-specific modifications, here is the place to do it.
if ("${CMAKE_C_COMPILER_ID}" STREQUAL "GNU")
    # cmake 2.8 workaround: gcc needs to be told to do ANSI C11.
    # cmake 3.0 doesn't have this problem.
    set ( CMAKE_C_FLAGS  "${CMAKE_C_FLAGS} -std=c11 -lm " )
    ...
elseif ("${CMAKE_C_COMPILER_ID}" STREQUAL "Intel")
    ...
elseif ("${CMAKE_C_COMPILER_ID}" STREQUAL "Clang")
    ...
elseif ("${CMAKE_C_COMPILER_ID}" STREQUAL "MSVC")
    ...
endif ( )
```

To compile SuiteSparse:GraphBLAS without running the demos, use
`make library` in the top-level directory, or `make` in the `build` directory.

Several compile-time options can be selected by editing the `Source/GB.h` file, but these are meant only for code development of SuiteSparse:GraphBLAS itself, not for end-users of SuiteSparse:GraphBLAS.

16.8 Using a plain makefile

The `GraphBLAS/alternative` directory contains a simple `Makefile` that can be used to compile SuiteSparse:GraphBLAS. This is a useful option if you do not have the required version of `cmake`. This `Makefile` can even compile the entire library with a C++ compiler, which cannot be done with `CMake`.

This alternative `Makefile` does not build the `libgraphblas_renamed.so` library required for MATLAB R2021a (see Section 16.6). This can be done by revising the `Makefile`, however: add the `-DGBRENAME=1` flag, and change the library name from `libgraphblas` to `libgraphbas_renamed`.

16.9 Running the Demos

After `make` in the top-level directory to compile the library, type `make run` to run the demos. You can also run the demos after compiling:

```
cd Demo
./demo
```

The `./demo` command is a script that runs the demos with various input matrices in the `Demo/Matrix` folder. The output of the demos will be compared with expected output files in `Demo/Output`.

NOTE: DO NOT publish benchmarks of these demos, and do not link against the demo library in any user application. These codes are sometimes slow, and are meant as simple illustrations only, not for performance. The fastest methods are in LAGraph, not in SuiteSparse/GraphBLAS/Demo. Benchmark LAGraph instead. Eventually, all GraphBLAS/Demos methods will be removed, and LAGraph will serve all uses: for illustration, benchmarking, and production uses.

16.10 Installing SuiteSparse:GraphBLAS

To install the library (typically in `/usr/local/lib` and `/usr/local/include` for Linux systems), go to the top-level GraphBLAS folder and type:

```
sudo make install
```

16.11 Linking issues after installation

My Linux distro (Ubuntu 18.04) includes a copy of `libgraphblas.so.1`, which is SuiteSparse:GraphBLAS v1.1.2. After installing SuiteSparse:GraphBLAS in `/usr/local/lib` (with `sudo make install`), compiling a simple stand-alone program links against `libgraphblas.so.1` instead of the latest version, while at the same time accessing the latest version of the include file as `/usr/local/include/GraphBLAS.h`. This command fails:

```
gcc prog.c -lgraphblas
```

Revising my `LD_LIBRARY_PATH` to put `/usr/local/lib` first in the library directory order didn't help. If you encounter this problem, try one of the following options (all four work for me, and link against the proper version (`/usr/local/lib/libgraphblas.so.6.1.4` for example):

```
gcc prog.c -l:libgraphblas.so.6
gcc prog.c -l:libgraphblas.so.6.1.4
gcc prog.c /usr/local/lib/libgraphblas.so
gcc prog.c -Wl,-v -L/usr/local/lib -lgraphblas
```

This `prog.c` test program is a trivial one, which works in v1.0 and later:

```
#include <GraphBLAS.h>
int main (void)
{
    GrB_init (GrB_NONBLOCKING) ;
    GrB_finalize ( ) ;
}
```

Compile the program above, then use this command to ensure `libgraphblas.so.6` appears:

```
ldd a.out
```

16.12 Running the tests

To run a short test, type `make run` at the top-level `GraphBLAS` folder. This will run all the demos in `GraphBLAS/Demos`. MATLAB is not required.

To perform the extensive tests in the `Test` folder, and the statement coverage tests in `Tcov`, MATLAB R2017A is required. See the `README.txt` files in those two folders for instructions on how to run the tests. The tests in the `Test` folder have been ported to MATLAB on Linux, MacOS, and Windows. The `Tcov` tests do not work on Windows. The MATLAB interface test (`gbtest`) works on all platforms; see the `GraphBLAS/GraphBLAS` folder for more details.

16.13 Cleaning up

To remove all compiled files, type `make distclean` in the top-level GraphBLAS folder.

17 About NUMA systems

I have tested this package extensively on multicore single-socket systems, but have not yet optimized it for multi-socket systems with a NUMA architecture. That will be done in a future release. If you publish benchmarks with this package, please state the SuiteSparse:GraphBLAS version, and a caveat if appropriate. If you see significant performance issues when going from a single-socket to multi-socket system, I would like to hear from you so I can look into it.

18 Acknowledgments

I would like to thank Jeremy Kepner (MIT Lincoln Laboratory Supercomputing Center), and the GraphBLAS API Committee: Aydın Buluç (Lawrence Berkeley National Laboratory), Timothy G. Mattson (Intel Corporation) Scott McMillan (Software Engineering Institute at Carnegie Mellon University), José Moreira (IBM Corporation), Carl Yang (UC Davis), and Benjamin Brock (UC Berkeley), for creating the GraphBLAS specification and for patiently answering my many questions while I was implementing it.

I would like to thank Tim Mattson and Henry Gabb, Intel, Inc., for their collaboration and for the support of Intel.

I would like to thank Joe Eaton and Corey Nolet for their collaboration on the CUDA kernels (still in progress), and for the support of NVIDIA.

I would like to thank John Eaton for his collaboration on the integration with Octave 7.

I would like to thank Michel Pelletier for his collaboration and work on the pygraphblas interface, and Jim Kitchen and Erik Welch for their work on Anaconda's python interface.

I would like to thank Will Kimmerer for his collaboration and work on the Julia interface.

I would like to thank John Gilbert (UC Santa Barbara) for our many discussions on GraphBLAS, and for our decades-long conversation and collaboration on sparse matrix computations.

I would like to thank Sébastien Villemot (Debian Developer, http://sebastien.villemot.name) for helping me with various build issues and other code issues with GraphBLAS (and all of SuiteSparse) for its packaging in Debian Linux.

I would like to thank Gábor Szárnyas for porting the `@GrB` interface to Octave 7 on Apple Silicon.

I would like to thank Roi Lipman, Redis (https://redislabs.com), for our many discussions on GraphBLAS and for enabling its use in RedisGraph (https://redislabs.com/redis-enterprise/technology/redisgraph/), a graph database module for Redis. Based on SuiteSparse:GraphBLAS, RedisGraph is up 600x faster than the fastest graph databases

(https://youtu.be/9h3Qco_x0QE
https://redislabs.com/blog/new-redisgraph-1-0-achieves-600x-faster-performance-graph-databases/).

SuiteSparse:GraphBLAS was developed with support from NVIDIA, Intel, MIT Lincoln Lab, Redis, IBM, the National Science Foundation (1514406, 1835499), and Julia Computing.

19 Additional Resources

See http://graphblas.org for the GraphBLAS community page. See https://github.com/GraphBLAS/GraphBLAS-Pointers for an up-to-date list of additional resources on GraphBLAS, maintained by Gábor Szárnyas.

References

[ACD+20] Mohsen Aznaveh, Jinhao Chen, Timothy A. Davis, Bálint Hegyi, Scott P. Kolodziej, Timothy G. Mattson, and Gábor Szárnyas. Parallel GraphBLAS with OpenMP. In *CSC20, SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 2020. https://www.siam.org/conferences/cm/conference/csc20.

[BBM+21] B. Brock, A. Buluç, T. Mattson, S. McMillan, and J. Moreira. The GraphBLAS C API specification (v2.0). Technical report, 2021. http://graphblas.org/.

[BG08] A. Buluç and J. Gilbert. On the representation and multiplication of hypersparse matrices. In *IPDPS'80: 2008 IEEE Intl. Symp. on Parallel and Distributed Processing*, pages 1–11, April 2008. https://dx.doi.org/10.1109/IPDPS.2008.4536313.

[BG12] A. Buluç and J. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012. https://dx.doi.org/10.1137/110848244.

[BMM+17a] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652, May 2017. https://dx.doi.org/10.1109/IPDPSW.2017.117.

[BMM+17b] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. The GraphBLAS C API specification. Technical report, 2017. http://graphblas.org/.

[DAK19] T. A. Davis, M. Aznaveh, and S. Kolodziej. Write quick, run fast: Sparse deep neural network in 20 minutes of development time via SuiteSparse:GraphBLAS. In *IEEE HPEC'19*. IEEE, 2019. Grand Challenge Champion, for high performance. See http://www.ieee-hpec.org/.

[Dav06] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, 2006.

Provides a basic overview of many sparse matrix algorithms and a simple sparse matrix data structure. A series of 42 lectures are available on YouTube; see the link at http://faculty.cse.tamu.edu/davis/publications.html For the book, see https://dx.doi.org/10.1137/1.9780898718881

[Dav18] T. A. Davis. Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and K-truss. In *IEEE HPEC'18*. IEEE, 2018. Grand Challenge Innovation Award. See http://www.ieee-hpec.org/.

[Dav19] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), December 2019.

[Dav21] Timothy A. Davis. Algorithm 10xx: SuiteSparse:GraphBLAS: Parallel graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 2021.

[DRSL16] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016.

Abstract: Wilkinson defined a sparse matrix as one with enough zeros that it pays to take advantage of them. This informal yet practical definition captures the essence of the goal of direct methods for solving sparse matrix problems. They exploit the sparsity of a matrix to solve problems economically: much faster and using far less memory than if all the entries of a matrix were stored and took part in explicit computations. These methods form the backbone of a wide range of problems in computational science. A glimpse of the breadth of applications relying on sparse solvers can be seen in the origins of matrices in published matrix benchmark collections (Duff and Reid 1979a, Duff, Grimes and Lewis 1989a, Davis and Hu 2011). The goal of this survey article is to impart a working knowledge of the underlying theory and practice of sparse direct methods for solving linear systems and least-squares problems, and to provide an overview of the algorithms, data structures, and software available to solve these problems, so that the reader can both understand the methods and know how best to use them. DOI: https://dx.doi.org/10.1017/S0962492916000076

[Gus78] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, 1978. https://dx.doi.org/10.1145/355791.355796.

[Hig02] N. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002. https://dx.doi.org/10.1137/1.9780898718027.

[Kep17] J. Kepner. GraphBLAS mathematics. Technical report, 2017. http://www.mit.edu/∼kepner/GraphBLAS/GraphBLAS-Math-release.pdf.

[KG11] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. SIAM, Philadelphia, PA, 2011.

From the preface: Graphs are among the most important abstract data types in computer science, and the algorithms that operate on them are critical to modern life. Graphs have been shown to be powerful tools for modeling complex problems because of their simplicity and generality. Graph algorithms are one of the pillars of mathematics, informing research in such diverse areas as combinatorial optimization, complexity theory, and topology. Algorithms on graphs are applied in many ways in today's world—from Web rankings to metabolic networks, from finite element meshes to semantic graphs. The current exponential growth in graph data has forced a shift to parallel computing for executing graph algorithms. Implementing parallel graph algorithms and achieving good parallel performance have proven difficult. This book addresses these challenges by exploiting the well-known duality between a canonical representation of graphs as abstract collections of vertices and edges and a sparse adjacency matrix representation. This linear algebraic approach is widely accessible to scientists and engineers who may not be formally trained in computer science. The authors show how to leverage existing parallel matrix computation techniques and the large amount of software infrastructure that exists for these computations to implement efficient and scalable parallel graph algorithms. The benefits of this approach are reduced algorithmic complexity, ease of implementation, and improved performance. DOI: https://dx.doi.org/10.1137/1.9780898719918

[MDK+19] T. Mattson, T. A. Davis, M. Kumar, A. Buluç, S. McMillan, J. Moreira, and C. Yang. LAGraph: a community effort to collect graph algorithms built on top of the GraphBLAS. In *GrAPL'19: Workshop on Graphs, Architectures, Programming, and Learning*. IEEE, May 2019. https://hpc.pnl.gov/grapl/previous/2019, part of IPDPS'19, at http://www.ipdps.org/ipdps2019.

[NMAB18] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. High-performance sparse matrix-matrix products on intel knl and multicore architectures. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[Wat87] A. J. Wathen. Realistic eigenvalue bounds for the Galerkin mass matrix. *IMA J. Numer. Anal.*, 7:449–457, 1987. https://dx.doi.org/10.1093/imanum/7.4.449.