User Guide for SuiteSparse:GraphBLAS

Timothy A. Davis

davis@tamu.edu, Texas A&M University.

http://suitesparse.com and http://aldenmath.com

VERSION 4.0.0, Nov 4, 2020 (DRAFT: DO NOT BENCHMARK; debug/etc on)

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 graph algorithms based on the elegant mathematics of sparse matrix operations on a semiring.

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, 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, and Carl Yang [BMM+17a, BMM+17b], 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 1.3.0 (Sept 25, 2019) 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.

> **SPEC:** See the tag **SPEC:** for SuiteSparse extensions to the spec. They are also placed in text boxes like this one. All functions, objects, and macros with a name of the form `GxB_*` are extensions to the spec.

1.1 Future plans

- Version 4.0.0 (likely in July, 2020), will follow the V2.0 of the C API. **The following is tentative, but all of these changes have been implemented in the current draft of v4.0.0 (July 1, 2020).**

  `GrB_wait()`, with no inputs is removed. `GrB_wait(&object)`: polymorphic function is added. `GrB_*_nvals` and related functions: no longer guarantee completion (per the v1.3 C API); use `GrB_wait(&object)` or non-polymorphic `GrB_*_wait(&object)` instead.

  `GrB_error` has changed; it now takes two parameters, `GrB_error(&s,C)` where `s` is the error string generated when `C` was last operated on.

  V4.0.0 is otherwise identical to V3.3.1.

1.2 Release Notes

- 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 spec. MATLAB interface: added complex matrices and operators, and changed behavior of integer operations to more closely match the behavior on MATLAB 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 descriptor, `GrB_DESC_*`. Added the structural mask option. Changed default chunk size to 65,536. Note that v3.2.0 is not compatible with the MS Visual Studio compiler; use v3.1.2 instead. MATLAB 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.

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

- Version 3.1.0 (Oct 1, 2019). MATLAB 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 5.10), in support of the new LAGraph project (https://github.com/GraphBLAS/LAGraph, see also Section 11.1). 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 10). PageRank demos have been added to the `Demos` folder. Prior versions required GraphBLAS to be compiled with OpenMP, for it to be thread-safe. It can now be compiled with POSIX pthreads. The `cmake` script automatically detects if OpenMP and/or POSIX pthreads are available. Demos have been added to show how GraphBLAS can be called from a multi-threaded user application.

- Version 2.1 (Oct 2018) was a major update with support for new matrix formats (by row or column, and hypersparse matrices), and MATLAB-like 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 5.11, 7, and 8.

- 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.

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 the MATLAB language, created by Cleve Moler. 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. 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 sup-

ports two kinds of sparse matrix formats: a regular sparse format, taking $O(n + |\mathbf{A}|)$ space, and a hypersparse format taking only $O(|\mathbf{A}|)$ space. As a result, 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 $|\mathbf{A}|$.

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* 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 it cannot be done in GraphBLAS. 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, sparse scalars, types, operators, monoids, semirings, and a descriptor object used for parameter settings. Details are given in Section 5. Once these objects are created they can be used in mathematical *operations* (not to be confused with the how the term *oper-*

ator is used in GraphBLAS). A short summary of these operations and their nearest MATLAB analog is given in the table below.

operation	approximate 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`

GraphBLAS can do far more than what 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 9. With this brief overview, the full scope of GraphBLAS extensions of these operations can now be described.

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. 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 2,438 unique semirings. With typecasting, any of these 2,438 semirings can be applied to matrices `C`, `A`, and `B` of 13 predefined types, in any combination. This gives over 5 million 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 5.11) 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 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

```
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, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$

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; these are discussed in the `GraphBLAS_Test.pdf` document in the `GraphBLAS/Test` folder.

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 4.5 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 states that typecasting follows the rules of ANSI C. Yet C leaves some typecasting undefined. SuiteSparse:GraphBLAS provides a precise definition for all typecasting as an extension to the spec.

2.5 Notation and list of GraphBLAS operations

As a summary of what GraphBLAS can do, the following table lists all GraphBLAS operations (where `GxB_*` are in SuiteSparse:GraphBLAS only). Upper case letters denote a matrix, lower case letters are vectors, and **AB** denote the multiplication of two matrices over a semiring.

`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	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`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	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, y)$
		$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(x, \mathbf{A})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, y)$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(x, \mathbf{u})$
`GxB_select`	apply select operator	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, k)$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, k)$
`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	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathrm{kron}(\mathbf{A}, \mathbf{B})$

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.

3 Interfaces to MATLAB, Python, Julia, Java

The 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.

A beta version of a Python interface is 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 MATLAB Interface

As of Version 3.1, a MATLAB interface is now available. Refer to 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 MATLAB interface adds the `GrB` class, which is an opaque MATLAB object that contains a GraphBLAS matrix, either double or single precision (real or complex), boolean, or any of the built-in integer types. 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 MATLAB, or `help GrB` for more details.

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

- Saving a GrB matrix object from MATLAB can be done, but the resulting `*.mat` file must be read in by the same version of GraphBLAS.
- `GrB` matrices with dimension larger than `2^53` do not display properly in the MATLAB `whos` command. MATLAB gets this information from `size(A)`, which returns a correct result, but MATLAB rounds it to double before displaying it. The size is displayed correctly with `disp` or `display`.

- Non-blocking mode is not exploited; this would require a MATLAB mexFunction to modify its inputs, which is technically possible but not permitted by the MATLAB API. This can have significant impact on performance, if a MATLAB m-file makes many repeated tiny changes to a matrix. This kind of computation can often be done with good performance in the C API, but will be very slow in MATLAB.
- Linear indexing, or `A(:)` for a 2D matrix, and a single output of `I=find(A)`.
- The second output for `min` and `max`, and the `includenan` option.
- 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. Saturating binary operators could be added in the future, so that `GrB.eadd (A, '+saturate', B)` could return the MATLAB result.
- Solvers, so that `x=A\b` could return a GF(2) solution, for example.
- Sparse matrices with dimension higher than 2. It would be possible to map an N-dimensional matrix to a large 2D hypersparse GraphBLAS matrix.

3.2 Python Interface

See Michel Pelletier's Python interface at https://github.com/michelp/pygraphblas. Anaconda is also developing a Python interface to SuiteSparse:GraphBLAS.

3.3 Julia Interface

See Abhinav Mehndiratta's Julia interface at https://github.com/abhinavmehndiratta/SuiteSparseGraphBLAS.jl.

3.4 Java Interface

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

4 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_` denote items that appear in the official *GraphBLAS C API Specification*. The prefix `GxB_` refers to SuiteSparse-specific extensions to the GraphBLAS API. Both may be used in user applications but be aware that items with prefixes `GxB_` will not appear in other implementations of the GraphBLAS standard.

SPEC: The following macros are extensions to the spec.

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 (1,4,0)
... use features from version 1.4.0 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. The SuiteSparse:GraphBLAS name is provided in its `GraphBLAS.h` file as:

```
#define GxB_SUITESPARSE_GRAPHBLAS
```

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 create, modify, and destroy the GraphBLAS context, or provide utility methods for dealing with errors:

GraphBLAS function	purpose	Section
`GrB_init`	start up GraphBLAS	4.1
`GrB_getVersion`	C API supported by the library	4.2
`GxB_init`	start up GraphBLAS with different `malloc`	4.3
`GrB_Info`	status code returned by GraphBLAS functions	4.4
`GrB_error`	get more details on the last error	4.5
`GrB_finalize`	finish GraphBLAS	4.6

4.1 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 to the user application. 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_*_nvals`, `GrB_*_extractElement`, `GrB_*_extractTuples`, and `GrB_*_reduce` (to scalar). These functions do *not* guarantee that the opaque objects they depend on are finalized. To do that, use `GrB_wait(&object)` instead.

SuiteSparse:GraphBLAS is multithreaded internally, via OpenMP, and it is also safe to use in a multithreaded user application. See Section 12 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_Matrix_wait` or `GrB_Vector_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(&object)` 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.

You can query the mode of a GraphBLAS session with the following (see Section 7), which returns the `mode` passed to `GrB_init`:

```
GrB_mode mode ;
GxB_get (GxB_MODE, &mode) ;
```

4.2 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.

This version of SuiteSparse:GraphBLAS supports 1.3.0 (Sept 25, 2019) of the C API Specification.

4.3 GxB_init: initialize with alternate malloc

```
GrB_Info GxB_init           // start up GraphBLAS and also define malloc, etc
(
    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 user_malloc_is_thread_safe
) ;
```

`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_*import` and `GxB_*export` functions described in Section 5.10, since they require the user application and GraphBLAS to use the same memory manager.

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 last argument to `GxB_init` informs GraphBLAS as

to whether or not the functions are thread-safe. The ANSI C and Intel TBB functions are thread-safe, but the MATLAB `mxMalloc` and related functions are not thread-safe. If not thread-safe, GraphBLAS calls the functions from inside an OpenMP critical section.

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

```
GxB_init (mode, malloc, calloc, realloc, free, true) ;
```

SuiteSparse:GraphBLAS can be compiled as normal (outside of MATLAB) and then linked into a MATLAB `mexFunction`. However, a `mexFunction` should use the MATLAB memory managers. To do this, use the following instead of `GrB_init(mode)` in a MATLAB `mexFunction`, with the flag `false` since these functions are not thread-safe:

```
#include "mex.h"
#include "GraphBLAS.h"
...
GxB_init (mode, mxMalloc, mxCalloc, mxRealloc, mxFree, false) ;
```

Passing in the last parameter as `false` requires that GraphBLAS be compiled with OpenMP. Internally, SuiteSparse:GraphBLAS never calls any memory management function inside a parallel region. Results are undefined if all three of the following conditions hold: (1) the user application calls GraphBLAS in parallel from multiple user-level threads, (2) the memory functions are not thread-safe, and (3) GraphBLAS is not compiled with OpenMP. Safety is guaranteed if at least one of those conditions is false.

To use the scalable Intel TBB memory manager:

```
#include "tbb/scalable_allocator.h"
#include "GraphBLAS.h"
...
GxB_init (mode, scalable_malloc, scalable_calloc, scalable_realloc,
    scalable_free, true) ;
```

SPEC: `GxB_init` is an extension to the spec.

4.4 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.

`GrB_SUCCESS`	0	the method or operation was successful
`GrB_NO_VALUE`	1	the method was successful, but the entry does not appear in the matrix or vector. Its value is implicit.
`GrB_UNINITIALIZED_OBJECT`	2	object has not been initialized
`GrB_INVALID_OBJECT`	3	object is corrupted
`GrB_NULL_POINTER`	4	input pointer is `NULL`
`GrB_INVALID_VALUE`	5	generic error code; some value is bad
`GrB_INVALID_INDEX`	6	a row or column index is out of bounds; for indices passed as scalars, not in a list.
`GrB_DOMAIN_MISMATCH`	7	object domains are not compatible
`GrB_DIMENSION_MISMATCH`	8	matrix dimensions do not match
`GrB_OUTPUT_NOT_EMPTY`	9	output matrix already has values in it
`GrB_OUT_OF_MEMORY`	10	out of memory
`GrB_INSUFFICIENT_SPACE`	11	output array not large enough
`GrB_INDEX_OUT_OF_BOUNDS`	12	a row or column index is out of bounds; for indices in a list of indices.
`GrB_PANIC`	13	unrecoverable error.

Not all GraphBLAS methods or operations can return all status codes. Any GraphBLAS method or operation can return an out-of-memory condition, `GrB_OUT_OF_MEMORY`, or a panic, `GrB_PANIC`. These two errors, and the `GrB_INDEX_OUT_OF_BOUNDS` error, are called *execution errors*. The other errors are called *API* errors. An API error is detecting immediately, regardless of the blocking mode. The detection of an execution error may be deferred until the pending operations complete.

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].

4.5 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 fro `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 `GxB_SelectOp_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`, `GxB_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.

The `GrB_error` function is a polymorphic function for the following variants:

```
GrB_Info GrB_Type_error       (const char **error, const GrB_Type type) ;
GrB_Info GrB_UnaryOp_error    (const char **error, const GrB_UnaryOp op) ;
GrB_Info GrB_BinaryOp_error   (const char **error, const GrB_BinaryOp op) ;
GrB_Info GxB_SelectOp_error   (const char **error, const GxB_SelectOp op) ;
GrB_Info GrB_Monoid_error     (const char **error, const GrB_Monoid monoid) ;
GrB_Info GrB_Semiring_error   (const char **error, const GrB_Semiring semiring) ;
GrB_Info GxB_Scalar_error     (const char **error, const GxB_Scalar s) ;
GrB_Info GrB_Vector_error     (const char **error, const GrB_Vector v) ;
GrB_Info GrB_Matrix_error     (const char **error, const GrB_Vector A) ;
GrB_Info GrB_Descriptor_error (const char **error, const GrB_Descriptor d) ;
```

Currently, only `GrB_Matrix_error`, `GrB_Vector_error`, `GxB_Scalar_error`, and `GrB_Descriptor_error` are able to return non-empty error strings. The latter can return an error string only from `GrB_Descriptor_set` and `GxB_set(d,...)`.

The only GraphBLAS methods (Section 5) that return an error string are `*setElement`, `*removeElement`, `GxB_Matrix_Option_set(A,...)`, `GxB_Vector_Option_set(v,...)`, `GrB_Descriptor_set`, and `GxB_Desc_set(d,...)`. All GraphBLAS operations discussed in Section 9 can return an error string in their input/output object, except for `GrB_reduce` when reducing to a scalar.

4.6 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`.

5 GraphBLAS Objects and their Methods

GraphBLAS defines eight different objects to represent matrices and vectors, their scalar data type (or domain), binary and unary operators on scalar types, monoids, semirings, and a *descriptor* object used to specify optional parameters that modify the behavior of a GraphBLAS operation. SuiteSparse:GraphBLAS adds two additional objects: a sparse scalar (`GxB_Scalar`), and an operator for selecting entries from a matrix or vector (`GxB_SelectOp`).

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
`GxB_SelectOp`	a select 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
`GxB_Scalar`	a sparse 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.

SPEC: `GxB_SelectOp` and `GxB_Scalar` are extensions to GraphBLAS.

5.1 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 (`#include <stdbool.h>` and `#include <stdint.h>`), and the classes in MATLAB, as listed in the following table.

MATLAB allows for `double complex` sparse matrices, but the `class(A)` for such a matrix is just `double`. MATLAB treats the complex types as properties of a class.

GraphBLAS type	C type	MATLAB class	description	range
`GrB_BOOL`	`bool`	`logical`	Boolean	true (1), false (0)
`GrB_INT8`	`int8_t`	`int8`	8-bit signed integer	-128 to 127
`GrB_INT16`	`int16_t`	`int16`	16-bit integer	-2^{15} to $2^{15}-1$
`GrB_INT32`	`int32_t`	`int32`	32-bit integer	-2^{31} to $2^{31}-1$
`GrB_INT64`	`int64_t`	`int64`	64-bit integer	-2^{63} to $2^{63}-1$
`GrB_UINT8`	`uint8_t`	`uint8`	8-bit unsigned integer	0 to 255
`GrB_UINT16`	`uint16_t`	`uint16`	16-bit unsigned integer	0 to $2^{16}-1$
`GrB_UINT32`	`uint32_t`	`uint32`	32-bit unsigned integer	0 to $2^{32}-1$
`GrB_UINT64`	`uint64_t`	`uint64`	64-bit unsigned integer	0 to $2^{64}-1$
`GrB_FP32`	`float`	`single`	32-bit IEEE 754	`-Inf` to `+Inf`
`GrB_FP64`	`double`	`double`	64-bit IEEE 754	`-Inf` to `+Inf`
`GxB_FC32`	`float complex`	`single` `~isreal(.)`	32-bit IEEE 754 complex	`-Inf` to `+Inf`
`GxB_FC64`	`double complex`	`double` `~isreal(.)`	64-bit IEEE 754 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. 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 5.1.1). 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:

`GrB_Type_new`	create a user-defined type
`GrB_Type_wait`	wait for a user-defined type
`GxB_Type_size`	return the size of a type
`GrB_Type_free`	free a user-defined type

5.1.1 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. For example, the user application can define its own type for complex numbers, but then transposing the matrix with GraphBLAS will not compute the complex conjugate transpose. This corresponds to the array transpose in MATLAB (`C=A.'`) instead of the complex conjugate transpose (`C=A'`). To compute the complex conjugate transpose, the application would need to create a user-defined unary operator to conjugate a user-defined complex scalar, and then apply it to the matrix before or after the transpose, via `GrB_apply`. An extensive set of complex operators are provided in the `usercomplex.c` example in the `Demo` folder, along with an include file, `usercomplex.h`, that is suitable for inclusion in any user application. GraphBLAS does not include any complex types or operators, SuiteSparse:GraphBLAS provides them in two simple "user" files in the `Demo` folder, as user-defined types. They also now appear as built-in types, `GxB_FC32` and `GxB_FC64`. Refer to Section 11.9 for more details on these example user-defined types.

5.1.2 GrB_Type_wait: wait for a type

```
GrB_Info GrB_wait               // wait for a user-defined type
(
    GrB_Type *type              // type to wait for
) ;
```

After creating a user-defined type, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. `GrB_Type_wait(&type)` ensures the `type` is completed. SuiteSparse:GraphBLAS currently does nothing for `GrB_Type_wait(&type)`, except to ensure that `type` is valid.

5.1.3 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)`.

SPEC: `GxB_Type_size` is an extension to the spec.

5.1.4 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 5.12 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`).

5.2 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 place-holder 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`), and Z refers to any complex floating point type (`FC32` or `FC64`).

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

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

`GxB_FREXPX` `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 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.

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	$Z \to F$	$z = \|x\|$	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

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 spec.

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

`GrB_UnaryOp_new`	create a user-defined unary operator
`GrB_UnaryOp_wait`	wait for a user-defined unary operator
`GxB_UnaryOp_ztype`	return the type of the output z for $z = f(x)$
`GxB_UnaryOp_xtype`	return the type of the input x for $z = f(x)$
`GrB_UnaryOp_free`	free a user-defined unary operator

5.2.1 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 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.

5.2.2 GrB_UnaryOp_wait: wait for a unary operator

```
GrB_Info GrB_wait               // wait for a user-defined unary operator
(
    GrB_UnaryOp *unaryop        // unary operator to wait for
) ;
```

After creating a user-defined unary operator, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. `GrB_UnaryOp_wait(&unaryop)` ensures the `op` is completed. SuiteSparse:GraphBLAS currently does nothing for `GrB_UnaryOp_wait(&unaryop)`, except to ensure that the `unaryop` is valid.

5.2.3 GxB_UnaryOp_ztype: return the type of z

```
GrB_Info GxB_UnaryOp_ztype          // return the type of z
(
    GrB_Type *ztype,                // return type of output z
    GrB_UnaryOp unaryop             // unary operator
) ;
```

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

SPEC: `GxB_UnaryOp_ztype` is an extension to the spec.

5.2.4 GxB_UnaryOp_xtype: return the type of x

```
GrB_Info GxB_UnaryOp_xtype          // return the type of x
(
    GrB_Type *xtype,                // return type of input x
    GrB_UnaryOp unaryop             // unary operator
) ;
```

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

SPEC: `GxB_UnaryOp_xtype` is an extension to the spec.

5.2.5 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.

5.3 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*` comparison operators 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 comparison operators 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
`GxB_PAIR_`T	$T \times T \to T$	$z = 1$	one
`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`, and relative comparisons 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 comparison operators 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 comparison operators are useful as "multiply" operators for creating semirings with Boolean monoids.

Binary comparison operators 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 comparison operators 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

Finally, 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 comparison operators 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 do the same comparison and produce a result with same type T as their input operands, returning one for true or zero for false. The `IS*` comparison operators are useful when combining comparisons with other non-Boolean operators. For example, a `PLUS-ISEQ` semiring counts how many terms of the comparison 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.

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

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 5.1 for a list of integer ranges.

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

`GrB_BinaryOp_new`	create a user-defined binary operator
`GrB_BinaryOp_wait`	wait for a user-defined binary operator
`GxB_BinaryOp_ztype`	return the type of the output z for $z = f(x, y)$
`GxB_BinaryOp_xtype`	return the type of the input x for $z = f(x, y)$
`GxB_BinaryOp_ytype`	return the type of the input y for $z = f(x, y)$
`GrB_BinaryOp_free`	free a user-defined binary operator

5.3.1 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.

5.3.2 GrB_BinaryOp_wait: wait for a binary operator

```
GrB_Info GrB_wait               // wait for a user-defined binary operator
(
    GrB_BinaryOp *binaryop      // binary operator to wait for
) ;
```

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

5.3.3 GxB_BinaryOp_ztype: return the type of z

```
GrB_Info GxB_BinaryOp_ztype         // return the type of z
(
    GrB_Type *ztype,                // return type of output z
    GrB_BinaryOp binaryop           // binary operator to query
) ;
```

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

SPEC: `GxB_BinaryOp_ztype` is an extension to the spec.

5.3.4 GxB_BinaryOp_xtype: return the type of x

```
GrB_Info GxB_BinaryOp_xtype         // return the type of x
(
    GrB_Type *xtype,                // return type of input x
    GrB_BinaryOp binaryop           // binary operator to query
) ;
```

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

SPEC: `GxB_BinaryOp_xtype` is an extension to the spec.

5.3.5 GxB_BinaryOp_ytype: return the type of y

```
GrB_Info GxB_BinaryOp_ytype         // return the type of y
(
    GrB_Type *ytype,                // return type of input y
    GrB_BinaryOp binaryop           // binary operator to query
) ;
```

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

SPEC: `GxB_BinaryOp_ytype` is an extension to the spec.

5.3.6 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.

5.3.7 ANY and PAIR operators

SuiteSparse:GraphBLAS v3.2.0 adds two new operators, `ANY` and `PAIR`.

The `PAIR` operator 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 symbolic 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 have to access the values of the matrix, so it is a very fast operator to use.

The `ANY` operator is very unusual, but very powerful. It is the function $f(x, y) = x$, 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 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 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.

The result of the `ANY` monoid is non-deterministic, unless it is coupled with the `PAIR` multiplicative operator. In this case, the `ANY_PAIR` semiring will return a deterministic result, since $f(1, 1)$ is always 1, for the `ANY` operator $f(x, y)$.

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.

5.4 SuiteSparse:GraphBLAS select operators: GxB_SelectOp

A select operator is a scalar function of the form $z = f(i, j, m, n, a_{ij}, \text{thunk})$ that is applied to the entries a_{ij} of an m-by-n matrix. The domain (type) of z is always boolean. The domain (type) of a_{ij} can be any built-in or user-defined type, or it can be `GrB_NULL` if the operator is type-generic.

The `GxB_SelectOp` operator is used by `GxB_select` (see Section 9.12) to select entries from a matrix. Each entry `A(i,j)` is evaluated with the operator, which returns true if the entry is to be kept in the output, or false if it is not to appear in the output. The signature of the select function `f` is as follows:

```
bool f                      // returns true if A(i,j) is kept
(
    const GrB_Index i,      // row index of A(i,j)
    const GrB_Index j,      // column index of A(i,j)
    const GrB_Index nrows,  // number of rows of A
    const GrB_Index ncols,  // number of columns of A
    const void *x,          // value of A(i,j), or NULL if f is type-generic
    const void *thunk       // user-defined auxiliary data
) ;
```

Operators can be used on any type, including user-defined types, except that the comparisons `GT`, `GE`, `LT`, and `LE` can only be used with built-in types. User-defined select operators can also be created.

GraphBLAS name	MATLAB analog	description
`GxB_TRIL`	`C=tril(A,k)`	true for `A(i,j)` if `(j-i) <= k`
`GxB_TRIU`	`C=triu(A,k)`	true for `A(i,j)` if `(j-i) >= k`
`GxB_DIAG`	`C=diag(A,k)`	true for `A(i,j)` if `(j-i) == k`
`GxB_OFFDIAG`	`C=A-diag(A,k)`	true for `A(i,j)` if `(j-i) != k`
`GxB_NONZERO`	`C=A(A~=0)`	true if `A(i,j)` is nonzero
`GxB_EQ_ZERO`	`C=A(A==0)`	true if `A(i,j)` is zero
`GxB_GT_ZERO`	`C=A(A>0)`	true if `A(i,j)` is greater than zero
`GxB_GE_ZERO`	`C=A(A>=0)`	true if `A(i,j)` is greater than or equal to zero
`GxB_LT_ZERO`	`C=A(A<0)`	true if `A(i,j)` is less than zero
`GxB_LE_ZERO`	`C=A(A<=0)`	true if `A(i,j)` is less than or equal to zero
`GxB_NE_THUNK`	`C=A(A~=k)`	true if `A(i,j)` is not equal to `k`
`GxB_EQ_THUNK`	`C=A(A==k)`	true if `A(i,j)` is equal to `k`
`GxB_GT_THUNK`	`C=A(A>k)`	true if `A(i,j)` is greater than `k`
`GxB_GE_THUNK`	`C=A(A>=k)`	true if `A(i,j)` is greater than or equal to `k`
`GxB_LT_THUNK`	`C=A(A<k)`	true if `A(i,j)` is less than `k`
`GxB_LE_THUNK`	`C=A(A<=k)`	true if `A(i,j)` is less than or equal to `k`

SPEC: `GxB_SelectOp` and the table above are extensions to the spec.

The following methods operate on the `GxB_SelectOp` object:

`GxB_SelectOp_new`	create a user-defined select operator
`GxB_SelectOp_wait`	wait for a user-defined select operator
`GxB_SelectOp_xtype`	return the type of the input x
`GxB_SelectOp_ttype`	return the type of the input *thunk*
`GxB_SelectOp_free`	free a user-defined select operator

5.4.1 GxB_SelectOp_new: create a user-defined select operator

```
GrB_Info GxB_SelectOp_new       // create a new user-defined select operator
(
    GxB_SelectOp *selectop,     // handle for the new select operator
    void *function,             // pointer to the select function
    GrB_Type xtype,             // type of input x, or NULL if type-generic
    GrB_Type ttype              // type of input thunk, or NULL if type-generic
) ;
```

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

The `function` argument to `GxB_SelectOp_new` is a pointer to a user-defined function whose signature is given at the beginning of Section 5.4. Given the properties of an entry a_{ij} in an m-by-n matrix, the `function` should return `true` if the entry should be kept in the output of `GxB_select`, or `false` if it should not appear in the output.

The type `xtype` is the GraphBLAS type of the input x of the user-defined function $z = f(i, j, m, n, x, \text{thunk})$. The type may be built-in or user-defined, or it may even be `GrB_NULL`. If the `xtype` is `GrB_NULL`, then the `selectop` is type-generic.

The type `ttype` is the GraphBLAS type of the input *thunk* of the user-defined function $z = f(i, j, m, n, x, \text{thunk})$. The type may be built-in or user-defined, or it may even be `GrB_NULL`. If the `ttype` is `GrB_NULL`, then the `selectop` does not access this parameter. The `const void *thunk` parameter on input to the user `function` will be passed as `NULL`.

5.4.2 GB_SelectOp_wait: wait for a select operator

```
GrB_Info GrB_wait               // wait for a user-defined select operator
(
    GxB_SelectOp *selectop      // select operator to wait for
) ;
```

After creating a user-defined select operator, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. `GxB_SelectOp_wait(&selectop)` ensures the `selectop` is completed. SuiteSparse:GraphBLAS currently does nothing for `GxB_SelectOp_wait(&selectop)`, except to ensure that the `selectop` is valid.

5.4.3 GxB_SelectOp_xtype: return the type of x

```
GrB_Info GxB_SelectOp_xtype     // return the type of x
(
    GrB_Type *xtype,            // return type of input x
    GxB_SelectOp selectop       // select operator
) ;
```

`GxB_SelectOp_xtype` returns the `xtype` of the select operator, which is the type of x in the function $z = f(i, j, m, n, x, \text{thunk})$. If the select operator is type-generic, `xtype` is returned as `GrB_NULL`. This is not an error condition, but simply indicates that the `selectop` is type-generic.

5.4.4 GxB_SelectOp_ttype: return the type of the *thunk*

```
GrB_Info GxB_SelectOp_ttype     // return the type of thunk
(
    GrB_Type *ttype,            // return type of input thunk
    GxB_SelectOp selectop       // select operator
) ;
```

`GxB_SelectOp_ttype` returns the `ttype` of the select operator, which is the type of *thunk* in the function $z = f(i, j, m, n, x, \text{thunk})$. If the select operator does not use this parameter, `ttype` is returned as `GrB_NULL`. This is not an error condition, but simply indicates that the `selectop` does not use this parameter.

5.4.5 GxB_SelectOp_free: free a user-defined select operator

```
GrB_Info GrB_free               // free a user-created select operator
(
    GxB_SelectOp *selectop      // handle of select operator to free
) ;
```

`GxB_SelectOp_free` frees a user-defined select operator. Either usage:

```
GxB_SelectOp_free (&selectop) ;
GrB_free (&selectop) ;
```

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

5.5 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 $T \times T \rightarrow T$). The associative 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.

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.

GraphBLAS operator	types (domains)	expression $z = f(x, y)$	identity	terminal
`GrB_PLUS_`T	$T \times T \rightarrow T$	$z = x + y$	0	none
`GrB_TIMES_`T	$T \times T \rightarrow T$	$z = xy$	1	0 (not F)
`GxB_ANY_`T	$T \times T \rightarrow T$	$z = x$ or y	any	any
`GrB_MIN_`R	$R \times R \rightarrow R$	$z = \min(x, y)$	$+\infty$	$-\infty$
`GrB_MAX_`R	$R \times R \rightarrow R$	$z = \max(x, y)$	$-\infty$	$+\infty$
`GrB_LOR`	`bool` $\times$ `bool` $\rightarrow$ `bool`	$z = x \vee y$	false	true
`GrB_LAND`	`bool` $\times$ `bool` $\rightarrow$ `bool`	$z = x \wedge y$	true	false
`GrB_LXOR`	`bool` $\times$ `bool` $\rightarrow$ `bool`	$z = x \veebar y$	false	none
`GrB_LXNOR`	`bool` $\times$ `bool` $\rightarrow$ `bool`	$z = (x == y)$	true	none
`GrB_BOR_`I	$I \times I \rightarrow I$	`z=x|y`	all bits zero	all bits one
`GrB_BAND_`I	$I \times I \rightarrow I$	`z=x&y`	all bits one	all bits zero
`GrB_BXOR_`I	$I \times I \rightarrow I$	`z=x^y`	all bits zero	none
`GrB_BXNOR_`I	$I \times I \rightarrow I$	`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 5.5.3 for an example.

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

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:

`GrB_Monoid_new`	create a user-defined monoid
`GrB_Monoid_wait`	wait for a user-defined monoid
`GxB_Monoid_terminal_new`	create a monoid that has a terminal value
`GxB_Monoid_operator`	return the monoid operator
`GxB_Monoid_identity`	return the monoid identity value
`GxB_Monoid_terminal`	return the monoid terminal value (if any)
`GrB_Monoid_free`	free a monoid

SPEC: The predefined `GxB*` monoids are an extension to the spec.

5.5.1 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 5.1), 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.

5.5.2 GrB_Monoid_wait: wait for a monoid

```
GrB_Info GrB_wait               // wait for a user-defined monoid
(
    GrB_Monoid *monoid          // monoid to wait for
) ;
```

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

5.5.3 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 5.5.1 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 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. 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`.

SPEC: `GxB_Monoid_terminal_new` is an extension to the spec.

5.5.4 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.

SPEC: `GxB_Monoid_operator` is an extension to the spec.

5.5.5 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.

SPEC: `GxB_Monoid_identity` is an extension to the spec.

5.5.6 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.

SPEC: `GxB_Monoid_terminal` is an extension to the spec.

5.5.7 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.

5.6 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:

`GrB_Semiring_new`	create a user-defined semiring
`GrB_Semiring_wait`	wait for a user-defined semiring
`GxB_Semiring_add`	return the additive monoid of a semiring
`GxB_Semiring_multiply`	return the binary operator of a semiring
`GrB_Semiring_free`	free a semiring

5.6.1 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. Using built-in types and operators, 2,438 semirings can be built. This count excludes redundant Boolean operators (for example `GrB_TIMES_BOOL` and `GrB_LAND` are different operators but they are redundant since they always return the same result).

The v1.3 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 1,473 of the 2,438 unique semirings that can be constructed from built-in types and operators, listed below, as an extension to the spec. 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 add monoids (MIN, MAX, PLUS, TIMES, ANY)
  - 20 multiply operators (FIRST, SECOND, PAIR, 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 comparison operator $T \times T \to$ bool, where T is non-Boolean and real, from the complete cross product of:
  - 5 Boolean add 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 add monoids (LAND, LOR, LXOR, EQ, ANY)
  - 11 multiply operators (FIRST, SECOND, PAIR, 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`, `PLUS`, `MINUS`, `TIMES`, `DIV`, `RDIV`, `RMINUS`)
  - 2 complex types, Z

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

SPEC: Predefined `GxB*` semirings are an extension to the spec.

5.6.2 GrB_Semiring_wait: wait for a semiring

```
GrB_Info GrB_wait               // wait for a user-defined semiring
(
    GrB_Semiring *semiring      // semiring to wait for
) ;
```

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

5.6.3 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.

SPEC: `GxB_Semiring_add` is an extension to the spec.

5.6.4 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.

SPEC: `GxB_Semiring_multiply` is an extension to the spec.

5.6.5 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.

5.7 GraphBLAS scalars: GxB_Scalar

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

SPEC: `GxB_Scalar` is an extension to the spec.

`GxB_Scalar_new`	create a sparse scalar
`GxB_Scalar_wait`	wait for a scalar
`GxB_Scalar_dup`	copy a sparse scalar
`GxB_Scalar_clear`	clear a sparse scalar of its entry
`GxB_Scalar_nvals`	return the number of entries in a sparse scalar (0 or 1)
`GxB_Scalar_type`	return the type of a sparse scalar
`GxB_Scalar_setElement`	set the single entry of a sparse scalar
`GxB_Scalar_extractElement`	get the single entry from a sparse scalar
`GxB_Scalar_free`	free a sparse scalar

5.7.1 GxB_Scalar_new: create a sparse scalar

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

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

5.7.2 GxB_Scalar_wait: wait for a scalar

```
GrB_Info GrB_wait               // wait for a scalar
(
    GxB_Scalar *s               // scalar to wait for
) ;
```

In non-blocking mode, the computations for a `GxB_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 `GxB_Scalar_wait(&s)`. After this call, different user threads may safely call GraphBLAS operations that use the scalar `s` as an input parameter.

5.7.3 GxB_Scalar_dup: copy a sparse scalar

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

`GxB_Scalar_dup` makes a deep copy of a sparse scalar, like `s=t` in MATLAB. In GraphBLAS, it is possible, and valid, to write the following:

```
GxB_Scalar t, s ;
GxB_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 sparse scalars are needed, then this should be used instead:

```
GxB_Scalar t, s ;
GxB_Scalar_new (&t, GrB_FP64) ;
GxB_Scalar_dup (&s, t) ;        // like s = t, but making a deep copy
```

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

5.7.4 GxB_Scalar_clear: clear a sparse scalar of its entry

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

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

5.7.5 GxB_Scalar_nvals: return the number of entries in a sparse scalar

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

`GxB_Scalar_nvals` returns the number of entries in a sparse scalar, which is either 0 or 1. Roughly analogous to `nvals = nnz(s)` 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.

5.7.6 GxB_Scalar_type: return the type of a sparse scalar

```
GrB_Info GxB_Scalar_type    // get the type of a GxB_Scalar
(
    GrB_Type *type,         // returns the type of the GxB_Scalar
    const GxB_Scalar s      // GxB_Scalar to query
) ;
```

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

5.7.7 GxB_Scalar_setElement: set the single entry of a sparse scalar

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

`GxB_Scalar_setElement` sets the single entry in a sparse scalar, like `s = sparse(x)` in MATLAB notation. For further details of this function, see `GxB_Matrix_setElement` in Section 5.9.10. If an error occurs, `GrB_error(&err,s)` returns details about the error.

5.7.8 GxB_Scalar_extractElement: get the single entry from a sparse scalar

```
GrB_Info GxB_Scalar_extractElement  // x = s
(
    <type> *x,                      // user scalar extracted
    const GxB_Scalar s              // GxB_Sclar to extract an entry from
) ;
```

`GxB_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 5.9.11, which discusses `GrB_Matrix_extractElement`. **NOTE:** if no entry is present in the sparse scalar `s`, then `x` is not modified, and the return value of `GxB_Scalar_extractElement` is `GrB_NO_VALUE`.

5.7.9 GxB_Scalar_free: free a sparse scalar

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

`GxB_Scalar_free` frees a sparse scalar. Either usage:

```
GxB_Scalar_free (&s) ;
GrB_free (&s) ;
```

frees the sparse 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 sparse scalar are abandoned.

5.8 GraphBLAS vectors: GrB_Vector

Many of the methods for GraphBLAS vectors require a row index or a size. Many methods for matrices require both a row and column index, or a row and column dimension. These are all integers of a specific type, `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 MATLAB. In SuiteSparse:GraphBLAS, the largest size permitted for any integer of `GrB_Index` is 2^{60}. The largest `GrB_Matrix` that SuiteSparse:GraphBLAS can construct is thus 2^{60}-by-2^{60}. An n-by-n matrix 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 7.3). The largest `GrB_Vector` that can be constructed is 2^{60}-by-1.

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

`GrB_Vector_new`	create a vector
`GrB_Vector_wait`	wait for a vector
`GrB_Vector_dup`	copy a vector
`GrB_Vector_clear`	clear a vector of all entries
`GrB_Vector_size`	return the size of a vector
`GrB_Vector_nvals`	return the number of entries in a vector
`GxB_Vector_type`	return the type of a vector
`GrB_Vector_build`	build a vector from a set of tuples
`GrB_Vector_setElement`	add a single entry to a vector
`GrB_Vector_extractElement`	get a single entry from a vector
`GrB_Vector_removeElement`	remove a single entry from a vector
`GrB_Vector_extractTuples`	get all entries from a vector
`GrB_Vector_resize`	resize a vector
`GrB_Vector_free`	free a vector
`GxB_Vector_import`	import a vector (see Section 5.10)
`GxB_Vector_export`	export a vector (see Section 5.10)

5.8.1 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 MATLAB statement `v = sparse (n,1)`, except that GraphBLAS can create sparse vectors any type. The pattern of the new vector is empty.

5.8.2 GrB_Vector_wait: wait for a vector

```
GrB_Info GrB_wait               // wait for a vector
(
    GrB_Vector *w               // vector to wait for
) ;
```

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)`. After this call, different user threads may safely call GraphBLAS operations that use the vector `w` as an input parameter.

5.8.3 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, like `w=u` in MATLAB. 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.

5.8.4 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.

5.8.5 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.

5.8.6 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.

5.8.7 GxB_Vector_type: return the type of a vector

```
GrB_Info GxB_Vector_type    // get the type of a vector
(
    GrB_Type *type,         // returns the type of the vector
    const GrB_Vector v      // vector to query
) ;
```

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

SPEC: `GxB_Vector_type` is an extension to the spec.

5.8.8 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 5.9.9), 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.

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

5.8.9 GrB_Vector_setElement: add a single 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 5.9.10. If an error occurs, `GrB_error(&err,w)` returns details about the error.

5.8.10 GrB_Vector_extractElement: get a single entry from a vector

```
GrB_Info GrB_Vector_extractElement  // x = v(i)
(
    <type> *x,                      // 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, so further details of this method are discussed in Section 5.9.11, which discusses `GrB_Matrix_extractElement`. In this case, the column index is implicitly `j=0`. **NOTE:** if no entry is present at `v(i)`, then `x` is not modified, and the return value of `GrB_Vector_extractElement` is `GrB_NO_VALUE`.

5.8.11 GrB_Vector_removeElement: remove a single 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.

5.8.12 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 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 5.9.13 where further details of this function are described.

5.8.13 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.

5.8.14 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.

5.9 GraphBLAS matrices: GrB_Matrix

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

`GrB_Matrix_new`	create a matrix
`GrB_Matrix_wait`	wait for a matrix
`GrB_Matrix_dup`	copy a matrix
`GrB_Matrix_clear`	clear a matrix of all entries
`GrB_Matrix_nrows`	return the number of rows of a matrix
`GrB_Matrix_ncols`	return the number of columns of a matrix
`GrB_Matrix_nvals`	return the number of entries in a matrix
`GxB_Matrix_type`	return the type of a matrix
`GrB_Matrix_build`	build a matrix from a set of tuples
`GrB_Matrix_setElement`	add a single entry to a matrix
`GrB_Matrix_extractElement`	get a single entry from a matrix
`GrB_Matrix_removeElement`	remove a single entry from a matrix
`GrB_Matrix_extractTuples`	get all entries from a matrix
`GrB_Matrix_resize`	resize a matrix
`GrB_Matrix_free`	free a matrix
`GxB_Matrix_import_CSR`	import a matrix in CSR form (see Section 5.10)
`GxB_Matrix_import_CSC`	import a matrix in CSC form (see Section 5.10)
`GxB_Matrix_import_HyperCSR`	import a matrix in HyperCSR form (see Section 5.10)
`GxB_Matrix_import_HyperCSC`	import a matrix in HyperCSC form (see Section 5.10)
`GxB_Matrix_export_CSR`	export a matrix in CSR form (see Section 5.10)
`GxB_Matrix_export_CSC`	export a matrix in CSC form (see Section 5.10)
`GxB_Matrix_export_HyperCSR`	export a matrix in HyperCSR form (see Section 5.10)
`GxB_Matrix_export_HyperCSC`	export a matrix in HyperCSC form (see Section 5.10)

5.9.1 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.

5.9.2 GrB_Matrix_wait: **wait for a matrix**

```
GrB_Info GrB_wait               // wait for a matrix
(
    GrB_Matrix *C               // matrix to wait for
) ;
```

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)`. After this call, different user threads may safely call GraphBLAS operations that use the matrix `C` as an input parameter.

5.9.3 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, like `C=A` in MATLAB. 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.

5.9.4 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.

5.9.5 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).

5.9.6 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).

5.9.7 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.

5.9.8 GxB_Matrix_type: return the type of a matrix

```
GrB_Info GxB_Matrix_type    // get the type of a matrix
(
\newpage
    GrB_Type *type,         // returns the type of the matrix
    const GrB_Matrix A      // matrix to query
) ;
```

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

SPEC: `GxB_Matrix_type` is an extension to the spec.

5.9.9 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.

SPEC: As an extension to the spec, 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.

5.9.10 GrB_Matrix_setElement: add a single 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.

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 9.10 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 11.4 for an example of how to use `GrB_Matrix_setElement`. If an error occurs, `GrB_error(&err,C)` returns details about the error.

5.9.11 GrB_Matrix_extractElement: get a single entry from a matrix

```
GrB_Info GrB_Matrix_extractElement      // x = A(i,j)
(
    <type> *x,                          // extracted 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`.

NOTE: if no entry is present at `A(i,j)`, then `x` is not modified, and the return value of `GrB_Matrix_extractElement` is `GrB_NO_VALUE`.

If the entry is not present then GraphBLAS does not know its value, since its value depends on the implicit value, which is the identity value of the additive monoid of the semiring. It is not a characteristic of the matrix itself, but of the semiring it is used in. A matrix can be used in any compatible semiring, and even a mixture of semirings, so the implicit value can change as the semiring changes.

As a result, 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 the entry is not present, `x` is not modified, and `GrB_NO_VALUE` is returned to the caller. What this means is up to the caller.

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 inter-

mixed with calls to these other two functions. Everything will work correctly and results will be predictable, it will just be slow.

5.9.12 GrB_Matrix_removeElement: remove a single 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.

5.9.13 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(&A)` 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.

5.9.14 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.

5.9.15 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.

5.10 GraphBLAS matrix and vector import/export

The import/export 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 semantics of import/export are the same as the *move constructor* in C++. On import, 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 import 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 import 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 an import, 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 export of a `GrB_Matrix` or `GrB_Vector` is symmetric with the import operation. The export changes the ownership of the arrays, where the `GrB_Matrix` or `GrB_Vector` no longer exists when the export completes, and instead the user is returned several arrays that 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-imported into a `GrB_Matrix` or `GrB_Vector`, at which point they again become the responsibility of GraphBLAS.

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

The vector import/export methods use a single format for a `GrB_Vector`. Four different formats are provided for the import/export of a `GrB_Matrix`. For each format, the `Ax` array 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 on import or export.

The table below lists the methods presented in this section.

method	purpose	Section
`GxB_Vector_import`	import a vector	5.10.1
`GxB_Vector_export`	export a vector	5.10.2
`GxB_Matrix_import_CSR`	import a matrix in CSR form	5.10.3
`GxB_Matrix_import_CSC`	import a matrix in CSC form	5.10.4
`GxB_Matrix_import_HyperCSR`	import a matrix in HyperCSR form	5.10.5
`GxB_Matrix_import_HyperCSC`	import a matrix in HyperCSC form	5.10.6
`GxB_Matrix_export_CSR`	export a matrix in CSR form	5.10.7
`GxB_Matrix_export_CSC`	export a matrix in CSC form	5.10.8
`GxB_Matrix_export_HyperCSR`	export a matrix in HyperCSR form	5.10.9
`GxB_Matrix_export_HyperCSC`	export a matrix in HyperCSC form	5.10.10

SPEC: The import/export methods are extensions to the spec. However, they have been implemented in SuiteSparse:GraphBLAS at the request of the GraphBLAS C API Committee, as a prototype for future consideration for inclusion in a future specification. Their calling sequence may change if these functions are added to the specification as `GrB_*` functions. A GraphBLAS library need not implement these methods in constant time and memory. On import, a library may choose to copy the content of the user arrays into its internal data structure and then `free` the user arrays. On export, it may chose to `malloc` the output arrays, fill them with the requested data, and then `GrB_free` the GraphBLAS object being exported. The semantics of these options are the same as a move constructor; they just take more time and memory. The choice is up to the GraphBLAS implementation since the internal data structure is opaque to the user application.

5.10.1 GxB_Vector_import: **import a vector**

```
GrB_Info GxB_Vector_import  // import a vector in CSC format
(
    GrB_Vector *v,          // vector to create
    GrB_Type type,          // type of vector to create
    GrB_Index n,            // vector length
    GrB_Index nvals,        // number of entries in the vector
    GrB_Index **vi,         // indices, size nvals (in sorted order)
    void      **vx,         // values, size nvals
    const GrB_Descriptor desc       // currently unused
) ;
```

The `GxB_Vector_import` function is a fast way to construct a `GrB_Vector`, always taking just $O(1)$ time. Calling `GxB_Vector_import` with:

```
GxB_Vector_import (&v, type, n, nvals, &vi, &vx, desc) ;
```

is identical to the following:

```
int64_t *Ap = calloc (2, sizeof (int64_t)) ;
Ap [1] = nvals ;
GxB_Matrix_import_CSC (&A, type, n, 1, nvals, -1, &Ap, &vi, &vx, desc) ;
```

except that the latter creates an `n`-by-1 matrix instead. For the vector import, described here, the first argument is a `GrB_Vector`. The arguments `vi` and `vx` take the place of `Ai` and `Ax`, and the `Ap` array for the CSC matrix import is not provided for a vector import. Refer to the description of `GxB_Matrix_import_CSC` for details (Section 5.10.4).

If successful, `v` is created as a `n`-by-1 vector. 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 the new vector `v`. The row indices in `vi` must appear in sorted order, and 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` is returned as `NULL` and `vi` and `vx` are not modified.

SPEC: `GxB_Vector_import` is an extension to the spec.

5.10.2 GxB_Vector_export: export a vector

```
GrB_Info GxB_Vector_export  // export and free a vector
(
    GrB_Vector *v,          // vector to export and free
    GrB_Type *type,         // type of vector exported
    GrB_Index *n,           // length of the vector
    GrB_Index *nvals,       // number of entries in the vector
    GrB_Index **vi,         // indices, size nvals
    void      **vx,         // values, size nvals
    const GrB_Descriptor desc       // currently unused
) ;
```

The `GxB_Vector_export` function is a fast way to extract the contents of a `GrB_Vector`, always taking just $O(1)$ time. Using `GxB_Vector_export` with:

```
GxB_Vector_export (&v, &type, &n, &nvals, &vi, &vx, desc) ;
```

is analogous to:

```
GxB_Matrix_export_CSC (&A, &type, &n, &one, &nvals, &nonempty,
    &Ap, &Ai, &Ax, desc)
```

if `A` were an `n`-by-1 matrix. For the vector export, described here, the first argument is a `GrB_Vector`. The arguments `vi` and `vx` take the place of `Ai` and `Ax`, and the `Ap` array for the CSC matrix export is not returned from a vector export. Refer to the description of `GxB_Matrix_export_CSC` for details. (Section 5.10.8).

Exporting a vector forces completion of any pending operations on the vector.

If successful, `v` is returned as `NULL`, and its contents are returned to the user, with its `type`, dimension `n`, and number of entries `nvals`. A sorted 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.

SPEC: `GxB_Vector_export` is an extension to the spec.

5.10.3 GxB_Matrix_import_CSR: import a CSR matrix

```
GrB_Info GxB_Matrix_import_CSR      // import a CSR matrix
(
    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_Index nvals,        // number of entries in the matrix
    // CSR format:
    int64_t nonempty,       // number of rows with at least one entry:
                            // either < 0 if not known, or >= 0 if exact
    GrB_Index **Ap,         // row "pointers", size nrows+1
    GrB_Index **Aj,         // column indices, size nvals
    void      **Ax,         // values, size nvals
    const GrB_Descriptor desc       // currently unused
) ;
```

`GxB_Matrix_import_CSR` imports a matrix from 3 user arrays in CSR format. In the resulting `GrB_Matrix A`, the `CSR` format is a matrix with a format (`GxB_FORMAT`) of `GxB_BY_ROW`, in standard for instead of hypersparse form (See Section 7.3).

The first four arguments of `GxB_Matrix_import_CSR` are the same as all four arguments of `GrB_Matrix_new`, because this function is similar. It creates a new `GrB_Matrix A`, with the given type and dimensions. The `GrB_Matrix A` does not exist on input.

Unlike `GrB_Matrix_new`, this function also populates the new 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, and must appear in sorted 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).

The `nonempty` parameter is optional. It states the number of rows that have at least one entry: if not known, use -1; if ≥ 0, it must be exact.

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:

```
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 [p] ;           // numerical value
    }
}
```

On successful creation of `A`, 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 the matrix `A` is later exported in CSR form, and GraphBLAS has not yet reallocated these arrays, then these same three arrays are returned to the user by `GxB_Matrix_export_CSR` (see Section 5.10.7). If an export is performed, the freeing of these three arrays again becomes the responsibility of the user application.

The `GxB_Matrix_import_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.

SPEC: `GxB_Matrix_import_CSR` is an extension to the spec.

5.10.4 GxB_Matrix_import_CSC: **import a CSC matrix**

```
GrB_Info GxB_Matrix_import_CSC      // import a CSC matrix
(
    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_Index nvals,        // number of entries in the matrix
    // CSC format:
    int64_t nonempty,       // number of columns with at least one entry:
                            // either < 0 if not known, or >= 0 if exact
    GrB_Index **Ap,         // column "pointers", size ncols+1
    GrB_Index **Ai,         // row indices, size nvals
    void      **Ax,         // values, size nvals
    const GrB_Descriptor desc       // currently unused
) ;
```

`GxB_Matrix_import_CSC` imports a matrix from 3 user arrays in CSC format. The `GrB_Matrix A` is created in the `CSC` format, which is a `GxB_FORMAT` of `GxB_BY_COL`. The arguments are identical to `GxB_Matrix_import_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 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 `nonempty` parameter is optional. It states the number of columns that have at least one entry: if not known, use -1; if ≥ 0, it must be exact.

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:

```
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 [p] ;           // numerical value
    }
}
```

The method is identical to `GxB_Matrix_import_CSR`; just the format is different. That is, if the method is successful, the 3 user arrays are imported into the new `GrB_Matrix A`, with the given type and dimensions, and returned as `NULL` pointers to the user application.

If `nvals` 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).

SPEC: `GxB_Matrix_import_CSC` is an extension to the spec.

5.10.5 GxB_Matrix_import_HyperCSR: import a HyperCSR matrix

```
GrB_Info GxB_Matrix_import_HyperCSR     // import a hypersparse CSR matrix
(
    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_Index nvals,        // number of entries in the matrix
    // hypersparse CSR format:
    int64_t nonempty,       // number of rows in Ah with at least one entry,
                            // either < 0 if not known, or >= 0 if exact
    GrB_Index nvec,         // number of rows in Ah list
    GrB_Index **Ah,         // list of size nvec of rows that appear in A
    GrB_Index **Ap,         // row "pointers", size nvec+1
    GrB_Index **Aj,         // column indices, size nvals
    void      **Ax,         // values, size nvals
    const GrB_Descriptor desc       // currently unused
) ;
```

`GxB_Matrix_import_HyperCSR` imports a matrix in hypersparse CSR format in $O(1)$ time. In the hypersparse format, 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 standard and hypersparse CSR formats. The row indices in `Ah` must appear in ascending order, and no duplicates can appear. To iterate over this data structure:

```
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 [p] ;           // numerical value
    }
}
```

This is more complex than the standard CSR format, but it requires at most $O(e)$ space, where A is m-by-n with $e =$ `nvals` entries. The standard 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 standard 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 import 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 created.

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). The `nonempty` parameter is optional. It states the number of rows that have at least one entry: if not known, use -1; if ≥ 0, it must be exact.

SPEC: `GxB_Matrix_import_HyperCSR` is an extension to the spec.

5.10.6 GxB_Matrix_import_HyperCSC: import a HyperCSC matrix

```
GrB_Info GxB_Matrix_import_HyperCSC     // import a hypersparse CSC matrix
(
    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_Index nvals,        // number of entries in the matrix
    // hypersparse CSC format:
    int64_t nonempty,       // number of columns in Ah with at least one entry,
                            // either < 0 if not known, or >= 0 if exact
    GrB_Index nvec,         // number of columns in Ah list
    GrB_Index **Ah,         // list of size nvec of columns that appear in A
    GrB_Index **Ap,         // column "pointers", size nvec+1
    GrB_Index **Ai,         // row indices, size nvals
    void      **Ax,         // values, size nvals
    const GrB_Descriptor desc       // currently unused
) ;
```

`GxB_Matrix_import_HyperCSC` imports a matrix in hypersparse CSC format in $O(1)$ time. It is identical to `GxB_Matrix_import_HyperCSR`, except for the data structure defined by the four arrays `Ah`, `Ap`, `Ai`, and `Ax`. It is a sparse array of size `nvec` of sparse column vectors. The following code iterates over the matrix:

```
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 [p] ;           // numerical value
    }
}
```

The `nonempty` parameter is optional. It states the number of columns that have at least one entry: if not known, use -1; if ≥ 0, it must be exact.

SPEC: `GxB_Matrix_import_HyperCSC` is an extension to the spec.

5.10.7 GxB_Matrix_export_CSR: export a CSR matrix

```
GrB_Info GxB_Matrix_export_CSR  // export and free a CSR matrix
(
    GrB_Matrix *A,          // handle of matrix to export and free
    GrB_Type *type,         // type of matrix exported
    GrB_Index *nrows,       // matrix dimension is nrows-by-ncols
    GrB_Index *ncols,
    GrB_Index *nvals,       // number of entries in the matrix
    // CSR format:
    int64_t *nonempty,      // number of rows with at least one entry
    GrB_Index **Ap,         // row "pointers", size nrows+1
    GrB_Index **Aj,         // column indices, size nvals
    void      **Ax,         // values, size nvals
    const GrB_Descriptor desc       // currently unused
) ;
```

`GxB_Matrix_export_CSR` exports a matrix in CSR form:

```
GxB_Matrix_export_CSR (&A, &type, &nrows, &ncols, &nvals, &nonempty,
                       &Ap, &Aj, &Ax, desc) ;
```

On successful output, the `GrB_Matrix A` is freed, and `A` is returned as `NULL`. Its type is returned in the `type` parameter, its dimensions in `nrows` and `ncols`, its number of entries in `nvals`, and the CSR format is in the three arrays `Ap`, `Aj`, and `Ax`. If `nvals` is zero, the `Aj` and `Ax` arrays are returned as `NULL`; this is not an error, and `GxB_Matrix_import_CSR` also allows these two arrays to be `NULL` on input when `nvals` is zero. After a successful export, 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 5.10.3.

This method takes $O(1)$ time if the matrix is already in standard (non-hypersparse) CSR format internally. If it is in hypersparse CSR form, the export must first convert the matrix to standard CSR form, taking $O(m)$ time and memory, where $m =$ `nrows`. If the matrix is in CSC format, it is first transposed to convert it to CSR format, and then exported. This takes $O(m + n + e)$ or $O(m + e \log e)$ time and memory, whichever is less, where $n =$ `ncols` and $e =$ `nvals`.

SPEC: `GxB_Matrix_export_CSR` is an extension to the spec.

5.10.8 GxB_Matrix_export_CSC: export a CSC matrix

```
GrB_Info GxB_Matrix_export_CSC  // export and free a CSC matrix
(
    GrB_Matrix *A,          // handle of matrix to export and free
    GrB_Type *type,         // type of matrix exported
    GrB_Index *nrows,       // matrix dimension is nrows-by-ncols
    GrB_Index *ncols,
    GrB_Index *nvals,       // number of entries in the matrix
    // CSC format:
    int64_t *nonempty,      // number of columns with at least one entry
    GrB_Index **Ap,         // column "pointers", size ncols+1
    GrB_Index **Ai,         // row indices, size nvals
    void      **Ax,         // values, size nvals
    const GrB_Descriptor desc       // currently unused
) ;
```

`GxB_Matrix_export_CSC` exports a matrix in CSC form:

```
GxB_Matrix_export_CSC (&A, &type, &nrows, &ncols, &nvals, &nonempty,
                       &Ap, &Ai, &Ax, desc) ;
```

On successful output, the `GrB_Matrix A` is freed, and `A` is returned as `NULL`. Its type is returned in the `type` parameter, its dimensions in `nrows` and `ncols`, its number of entries in `nvals`, and the CSC format is in the three arrays `Ap`, `Ai`, and `Ax`. If `nvals` is zero, the `Ai` and `Ax` arrays are returned as `NULL`; this is not an error, and `GxB_Matrix_import_CSC` also allows these two arrays to be `NULL` on input when `nvals` is zero. After a successful export, 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 5.10.4.

This method takes $O(1)$ time if the matrix is already in standard (non-hypersparse) CSC format internally. If it is in hypersparse CSC form, the export must first convert the matrix to standard CSC form, taking $O(n)$ time and memory, where $n =$ `ncols`. If the matrix is in CSR format, it is first transposed to convert it to CSC format, and then exported. This takes $O(m + n + e)$ or $O(n + e \log e)$ time and memory, whichever is less, where $m =$ `nrows` and $e =$ `nvals`.

SPEC: `GxB_Matrix_export_CSC` is an extension to the spec.

5.10.9 GxB_Matrix_export_HyperCSR: export a HyperCSR matrix

```
GrB_Info GxB_Matrix_export_HyperCSR  // export and free a hypersparse CSR matrix
(
    GrB_Matrix *A,          // handle of matrix to export and free
    GrB_Type *type,         // type of matrix exported
    GrB_Index *nrows,       // matrix dimension is nrows-by-ncols
    GrB_Index *ncols,
    GrB_Index *nvals,       // number of entries in the matrix
    // hypersparse CSR format:
    int64_t *nonempty,      // number of rows in Ah with at least one entry
    GrB_Index *nvec,        // number of rows in Ah list
    GrB_Index **Ah,         // list of size nvec of rows that appear in A
    GrB_Index **Ap,         // row "pointers", size nvec+1
    GrB_Index **Aj,         // column indices, size nvals
    void      **Ax,         // values, size nvals
    const GrB_Descriptor desc       // currently unused
) ;
```

`GxB_Matrix_export_HyperCSR` exports a matrix in CSR form:

```
GxB_Matrix_export_HyperCSR (&A, &type, &nrows, &ncols, &nvals, &nonempty,
                            &nvec, &Ah, &Ap, &Aj, &Ax, desc) ;
```

On successful output, the `GrB_Matrix A` is freed, and `A` is returned as `NULL`. Its type is returned in the `type` parameter, its dimensions in `nrows` and `ncols`, its number of entries in `nvals`, and the number of non-empty rows in `nvec`. The hypersparse CSR format is in the four arrays `Ah`, `Ap`, `Aj`, and `Ax`. If `nvals` is zero, the `Aj` and `Ax` arrays are returned as `NULL`; this is not an error. After a successful export, 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 5.10.5.

This method takes $O(1)$ time if the matrix is already in hypersparse CSR format internally. If it is in standard CSR form, the export must first convert the matrix to hypersparse CSR form, taking $O(m)$ time and memory, where $m =$ `nrows`. If the matrix is in CSC format, it is first transposed to convert it to hypersparse CSR format, and then exported. If in standard CSC form, the transpose takes $O(m + n + e)$ or $O(n + e \log e)$ time and memory, whichever is less. If in hypersparse CSC format, it takes $O(e \log e)$ time.

SPEC: `GxB_Matrix_export_HyperCSR` is an extension to the spec.

5.10.10 GxB_Matrix_export_HyperCSC: export a HyperCSC matrix

```
GrB_Info GxB_Matrix_export_HyperCSC  // export and free a hypersparse CSC matrix
(
    GrB_Matrix *A,          // handle of matrix to export and free
    GrB_Type *type,         // type of matrix exported
    GrB_Index *nrows,       // matrix dimension is nrows-by-ncols
    GrB_Index *ncols,
    GrB_Index *nvals,       // number of entries in the matrix
    // hypersparse CSC format:
    int64_t *nonempty,      // number of columns in Ah with at least one entry
    GrB_Index *nvec,        // number of columns in Ah list
    GrB_Index **Ah,         // list of size nvec of columns that appear in A
    GrB_Index **Ap,         // columns "pointers", size nvec+1
    GrB_Index **Ai,         // row indices, size nvals
    void      **Ax,         // values, size nvals
    const GrB_Descriptor desc       // currently unused
) ;
```

`GxB_Matrix_export_HyperCSC` exports a matrix in CSC form:

```
GxB_Matrix_export_HyperCSC (&A, &type, &nrows, &ncols, &nvals, &nonempty,
                            &nvec, &Ah, &Ap, &Ai, &Ax, desc) ;
```

On successful output, the `GrB_Matrix A` is freed, and `A` is returned as `NULL`. Its type is returned in the `type` parameter, its dimensions in `nrows` and `ncols`, its number of entries in `nvals`, and the number of non-empty rows in `nvec`. The hypersparse CSC format is in the four arrays `Ah`, `Ap`, `Ai`, and `Ax`. If `nvals` is zero, the `Ai` and `Ax` arrays are returned as `NULL`; this is not an error. After a successful export, 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 5.10.6.

This method takes $O(1)$ time if the matrix is already in hypersparse CSR format internally. If it is in standard CSR form, the export must first convert the matrix to hypersparse CSR form, taking $O(m)$ time and memory, where $m =$ `nrows`. If the matrix is in CSC format, it is first transposed to convert it to hypersparse CSR format, and then exported. If in standard CSC form, the transpose takes $O(m + n + e)$ or $O(n + e \log e)$ time and memory, whichever is less. If in hypersparse CSC format, it takes $O(e \log e)$ time.

SPEC: `GxB_Matrix_export_HyperCSC` is an extension to the spec.

5.11 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
}
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_HEAP      = 1002,   // heap-based saxpy method
    GxB_AxB_DOT       = 1003,   // dot product
    GxB_AxB_HASH      = 1004,   // hash-based saxpy method
    GxB_AxB_SAXPY     = 1005    // saxpy method (any kind)
}
GrB_Desc_Value ;
```

SPEC: `GxB_DEFAULT`, `GxB_NTHREADS`, `GxB_CHUNK`, `GxB_AxB_METHOD`, and `GxB_AxB_*` are extensions to the spec.

The internal representation is opaque to the user, but in this User Guide the five descriptor fields of a descriptor `desc` are illustrated as an array of five items, as described in the list below. The underlying implementation need not be an array:

- `desc [GrB_OUTP]` is a parameter that modifies the output of a GraphBLAS operation. Currently, there are two possible settings. In the default case, the output is not cleared, and $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ is 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.

- `desc [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 `desc [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 `desc [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 spec).

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.

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

  If the `desc [GrB_INP0]` is set to `GrB_TRAN`, then `A` is transposed before using it in the operation. Likewise, if `desc [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 `desc [GrB_INP0]` modifies the matrix parameter and `desc [GrB_INP1]` is ignored. If a method's first parameter is a vector or scalar and the second a matrix, then `desc [GrB_INP1]` modifies the matrix parameter and `desc [GrB_INP0]` is ignored.

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

- `desc [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. The current version always follows the hint, however.

  - `GxB_DEFAULT` means that a method is selected automatically.

– `GxB_AxB_SAXPY`: select any saxpy-based method: `GxB_AxB_GUSTAVSON`, `GxB_AxB_HEAP`, and/or `GxB_AxB_HASH`, or any mix of the three, 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)`). 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. 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, corresponding to the number of columns of `C`.

– `GxB_AxB_HEAP`: no longer appears in SuiteSparse:GraphBLAS, but may be reintroduced in a future version. This is silently replaced with `GxB_AxB_HASH`.

– `GxB_AxB_HASH`: a hash-based method, based on [NMAB18]. Very efficient for hypersparse matrices, matrix-vector-multiply, and when $|\mathbf{B}|$ is small.

– `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, or when `C` is tiny. 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.

5.11.1 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, and the method used in `C=A*B` is selected automatically).

5.11.2 GrB_Descriptor_wait: wait for a descriptor

```
GrB_Info GrB_wait                   // wait for a descriptor
(
    GrB_Descriptor *descriptor      // descriptor to wait for
) ;
```

After creating a user-defined descriptor, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. `GrB_Descriptor_wait(&d)` ensures the descriptor `d` is completed. SuiteSparse:GraphBLAS currently does nothing for `GrB_Descriptor_wait(&d)`, except to ensure that `d` is valid.

5.11.3 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 (`GxB_DEFAULT`, `GrB_COMP`, `GrB_STRUCTURE`, `GrB_COMP+GrB_STRUCTURE`, `GrB_TRAN`, `GrB_REPLACE`, `GxB_AxB_GUSTAVSON`, `GxB_AxB_HEAP`, `GxB_AxB_HASH`, `GxB_AxB_SAXPY`, or `GxB_AxB_DOT`).

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.
`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 used for computing `C=A*B` is selected automatically.	`GxB_AxB_`*method*: The selected method is used to compute `C=A*B`.

If an error occurs, `GrB_error(&err,desc)` returns details about the error.

5.11.4 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 descriptor settings that do not have the type `GrB_Desc_Value`. See also `GxB_set` described in Section 7. If an error occurs, `GrB_error(&err,desc)` returns details about the error.

SPEC: `GxB_Desc_set` is an extension to the spec.

5.11.5 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 7.

SPEC: `GxB_Desc_get` is an extension to the spec.

5.11.6 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.

5.11.7 GrB_DESC_*: predefined descriptors

Version 1.3 of the GraphBLAS C API Specification adds predefined descriptors, and these have been added as of v3.2.0 of SuiteSparse:GraphBLAS. They are listed in the table below. 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. `GrB_NULL` is the default descriptor, with all settings at their defaults: `OUTP`: do not replace the output, `MASK`: mask is valued and not complemented, `INP0`: first input not transposed, and `INP1`: second input not transposed.

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`

5.12 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`.

6 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 complemeted, 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.

repl	accum	**C**	**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

7 SuiteSparse:GraphBLAS Options

SPEC: GxB_set and GxB_get are extensions to the specification.

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)` provides hints to SuiteSparse:GraphBLAS on how it should store all matrices created after calling this function: by row, by column, and whether or not to use a *hypersparse* format [BG08, BG12]. These are global options that modify all matrices created after calling this method. The global settings also control the number of threads used, and the heuristic for selecting the number of threads for small problems.

- `GxB_set (GrB_Matrix A, field, value)` provides hints to SuiteSparse: GraphBLAS on how to store a particular matrix. This method allows SuiteSparse:GraphBLAS to transform a specific matrix from one format to another. The format has no effect on the result computed by GraphBLAS; it only affects the time and memory taken to do the computations.

- `GxB_set (GrB_Descriptor desc, field, value)` is another way to set the value of a field in a `GrB_Descriptor`. It is identical to `GrB_Descriptor_set`, just with a generic name.

The `GxB_get` method queries a `GrB_Descriptor`, a `GrB_Matrix`, or the global options.

- `GxB_get (field, &value)` retrieves the value of a global option.

- `GxB_get (GrB_Matrix A, field, &value)` retrieves the current value of an option from a particular matrix `A`.

- `GxB_get (GrB_Descriptor desc, field, &value)` retrieves the value of a field in a descriptor.

7.1 OpenMP parallelism

SuiteSparse:GraphBLAS Version 3 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.

7.2 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.

An example of this can be found in Section B.1 of Version 1.2 of the GraphBLAS API Specification, where the breadth-first search `BFS` uses `GrB_vxm` to compute `q'=q'*A`. This method is not fast if the matrix `A` is stored by column. The `bfs5` and `bfs6` examples in the `Demo/` folder of SuiteSparse:GraphBLAS use `GrB_vxm`, which is fast since the matrices are assumed to be stored in their default format, by row.

SuiteSparse:GraphBLAS stores its sparse matrices by row, by default. In Versions 2.1 and earlier, the matrices were stored by column, by default. 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. Alternatively, SuiteSparse:GraphBLAS can be compiled with `-DBYCOL`, which changes the default format to `GxB_BY_COL`, with no calls to any `GxB_*` function. The default format is now `GxB_BY_ROW`.

7.3 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`.

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
{
    GxB_HYPER_SWITCH = 0,  // defines switch to hypersparse (double value)
    GxB_FORMAT = 1,     // defines CSR/CSC format: GxB_BY_ROW or GxB_BY_COL
    GxB_MODE = 2,       // mode passed to GrB_init (blocking or non-blocking)
    GxB_GLOBAL_NTHREADS = GxB_NTHREADS, // max number of threads to use
    GxB_GLOBAL_CHUNK = GxB_CHUNK,       // chunk size for small problems

    TODO: add new options here
}
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 (in SuiteSparse:GraphBLAS Version 2.2 and later) is by row. The format in SuiteSparse:GraphBLAS Version 2.1 and earlier was by column, just like MATLAB.

The default format is given by the predefined value `GxB_FORMAT_DEFAULT`, which is equal to `GxB_BY_ROW` if default compile-time options are used. To change the default at compile time to `GxB_BY_COL`, compile the SuiteSparse: GraphBLAS library with `-DBYCOL`. This changes `GxB_FORMAT_DEFAULT` to `GxB_BY_COL`. 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_FORMAT_DEFAULT) ;
```

That is, by default, all new matrices are held by column in CSR format, unless `-DBYCOL` is used at compile time, in which case the default is to store all new matrices by row in CSC format. 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, wihch returns a value of `GxB_HYPERSPARSE` `GxB_SPARSE` `GxB_BITMAP` or `GxB_FULL`.

```
int sparsity ;
GxB_get (A, GxB_SPARSITY_STATUS, &sparsity) ;
```

7.4 Other global options

`GxB_MODE` can only be queried by `GxB_get`; they 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 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.

7.5 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_FORMAT`, `GxB_NTHREADS`, or `GxB_CHUNK`.

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, and the chunk size to 10000. 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) ;
```

7.6 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` 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`. 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.

```
GxB_set (A, GxB_HYPER_SWITCH, 0.2) ;
GxB_set (A, GxB_FORMAT, GxB_BY_COL) ;
```

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.

7.7 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`, or `GxB_CHUNK`. Refer to Sections 5.11.3 and 5.11.4 for details.

If an error occurs, `GrB_error(&err,desc)` returns details about the error.

7.8 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 `GxB_HYPER_SWITCH`, `GxB_FORMAT`. `GxB_MODE`, `GxB_NTHREADS`, or `GxB_CHUNK`. For example:

```
double h ;
GxB_get (GxB_HYPER_SWITCH, &h) ;
printf ("hyper_switch = %g for all new matrices\n", h) ;

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_NONBLOCK) 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) ;
```

7.9 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_SPARSITY_STATUS`, or `GxB_FORMAT`. For example:

```
double h ;
int sparsity ;
GxB_get (A, GxB_SPARSITY_STATUS, &sparsity) ;
GxB_get (A, GxB_HYPER_SWITCH, &h) ;
printf ("matrix A has hyper_switch = %g\n", h) ;
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") ;
```

7.10 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`, or `GxB_CHUNK`. Refer to Section 5.11.5 for details.

7.11 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) ;

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) ;
```

To get global options that can be queried but not modified:

```
GxB_get (GxB_MODE,          GrB_Mode *mode) ;
```

To set/get a matrix option:

```
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_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) ;
```

TODO: add SPARSITY here
To get the sparsity status of a matrix:

```
GxB_get (GrB_Matrix A, 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_HEAP) ;
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) ;
```

8 SuiteSparse:GraphBLAS Colon and Index Notation

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 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 9) 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 9.9 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`.

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 $\Delta =$ `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`.

SPEC: `GxB_RANGE`, `GxB_STRIDE`, and `GxB_BACKWARDS` are extensions to the specification.

9 GraphBLAS Operations

The next sections define each of the GraphBLAS operations, also listed in the table below. SuiteSparse:GraphBLAS extensions (`GxB_subassign`, `GxB_select`) are included in the table.

`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 union	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$
`GrB_eWiseAdd`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$
	set intersection	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`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	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(x, \mathbf{A})$
		$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, y)$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(x, \mathbf{x})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, y)$
`GxB_select`	apply select operator	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, \mathbf{k})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, \mathbf{k})$
`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	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathrm{kron}(\mathbf{A}, \mathbf{B})$

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`.

9.1 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.

9.2 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_Vector 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.

The breadth-first search presented in Section 11.2 of this User Guide uses `GrB_vxm` instead of `GrB_mxv`, since the default format in SuiteSparse:GraphBLAS is `GxB_BY_ROW`. If the matrix is stored by column, then use `GrB_mxv` instead.

9.3 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.

9.4 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.

9.4.1 GrB_eWiseMult_Vector: 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.

9.4.2 GrB_eWiseMult_Matrix: 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.

9.5 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:

for all entries (i, j) in $\mathbf{A} \cap \mathbf{B}$
    $t_{ij} = a_{ij} \oplus b_{ij}$
for all entries (i, j) in $\mathbf{A} \setminus \mathbf{B}$
    $t_{ij} = a_{ij}$
for all entries (i, j) in $\mathbf{B} \setminus \mathbf{A}$
    $t_{ij} = b_{ij}$

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.

9.5.1 GrB_eWiseAdd_Vector: 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.

9.5.2 GrB_eWiseAdd_Matrix: 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.

9.6 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.

9.6.1 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 8 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.

9.6.2 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 8 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.

9.6.3 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 8 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.

9.7 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 8 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 9.8. The two operations are compared and contrasted in Section 9.10. For a discussion of how duplicate indices are handled in `I` and `J`, see Section 9.9.

SPEC: All variants of `GxB_subassign` are extensions to the spec.

9.7.1 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 9.7.2, 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.

9.7.2 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.

9.7.3 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`.

9.7.4 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`.

9.7.5 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 9.7.6.

9.7.6 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`, 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 9.7.2 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 5.9.10). 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.

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.

9.8 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 9.7. They differ primarily in the size of the `Mask`, and how the `GrB_REPLACE` option works. Refer to Section 9.10 for a complete comparison of `GxB_subassign` and `GrB_assign`.

See Section 8 for a description of `I`, `ni`, `J`, and `nj`.

9.8.1 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 9.8.2, 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.

9.8.2 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.

9.8.3 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`.

9.8.4 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`.

9.8.5 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.

Following the C API Specification, results are well-defined if `I` contains duplicate indices. Duplicate indices are simply ignored. See Section 9.9 for more details.

9.8.6 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 9.8.2 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 5.9.10). 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.

Following the C API Specification, results are well-defined if `I` or `J` contain duplicate indices. Duplicate indices are simply ignored. See Section 9.9 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.

9.9 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 MATLAB interface to GraphBLAS to return the same results for `C(I,J)=A` for a `GrB` object as they do for built-in 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 spec. The spec only defines the behavior when assigning a scalar into a matrix or vector, and states that duplicate indices otherwise lead to undefined results.

9.10 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 2 and 3. 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 2 with Table 1 in Section 6.

The first column of Table 2 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 2: 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 3: Results of assign for entries outside the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix. Subassign has no effect on these entries.

Table 3 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 3. 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 3 differs from Table 2, 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 2, 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 3 only applies to entries in the single row `C(i,J)` that are outside the list of indices, `J`. For `GrB_Col_assign`, Table 3 only applies to entries in the single column `C(I,j)` that are outside the list of indices, `I`.

9.10.1 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`.

9.10.2 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 9.10. It can be up to 500x faster than MATLAB R2019b, or even higher depending on the kind of matrix assignment. MATLAB logical indexing (the mask of GraphBLAS) is much faster with GraphBLAS than in MATLAB R2019b; differences of up to 100,000x have been observed.

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 $h \log h$.

9.11 GrB_apply: apply a unary or binary operator

`GrB_apply` is the generic name for 62 specific functions. `GrB_Vector_apply` and `GrB_Matrix_apply` apply a unary operator to the entries of a matrix. `GrB_*_apply_BinaryOp1st*` applies a binary operator where a single scalar is provided as the x input to the binary operator. `GrB_*_apply_BinaryOp2nd*` applies a binary operator where a single scalar is provided as the y input to the binary operator. 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.

9.11.1 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.

9.11.2 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.

9.11.3 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. It is otherwise identical to `GrB_Vector_apply`. With no suffix, `GxB_Vector_apply_BinaryOp1st` takes as input a `GxB_Scalar`.

9.11.4 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. It is otherwise identical to `GrB_Vector_apply`. With no suffix, `GxB_Vector_apply_BinaryOp2nd` takes as input a `GxB_Scalar`.

9.11.5 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. It is otherwise identical to `GrB_Matrix_apply`. With no suffix, `GxB_Matrix_apply_BinaryOp1st` takes as input a `GxB_Scalar`. To transpose the input matrix, use the `GrB_INP0` descriptor setting.

9.11.6 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. It is otherwise identical to `GrB_Matrix_apply`. With no suffix, `GxB_Matrix_apply_BinaryOp2nd` takes as input a `GxB_Scalar`. To transpose the input matrix, use the `GrB_INP1` descriptor setting.

9.12 GxB_select: apply a select operator

The `GxB_select` function is the generic name for two specific functions: `GxB_Vector_select` and `GxB_Matrix_select`. 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 `GxB_select` is used.

> **SPEC:** The `GxB_select` operation and `GxB_SelectOp` operator are extensions to the spec.

9.12.1 GxB_Vector_select: apply a select operator to a vector

```
GrB_Info GxB_select                 // w<mask> = accum (w, op(u,k))
(
    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 GxB_SelectOp op,          // operator to apply to the entries
    const GrB_Vector u,             // first input:  vector u
    const GxB_Scalar Thunk,         // optional input for the select operator
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GxB_Vector_select` applies a select operator to the entries of a vector, analogous to `t = u.*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. If the operator is not type-generic, the entries in `u` are typecasted into the `xtype` of the select operator. The vector `t` has the same type and size as `u`. 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.

This operation operates on vectors just as if they were `m`-by-1 matrices, except that GraphBLAS never transposes a vector via the descriptor. The `op` is passed `n=1` as the number of columns. Refer to the next section on `GxB_Matrix_select` for more details.

9.12.2 GxB_Matrix_select: apply a select operator to a matrix

```
GrB_Info GxB_select                 // C<Mask> = accum (C, op(A,k)) or op(A',k)
(
    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 GxB_SelectOp op,          // operator to apply to the entries
    const GrB_Matrix A,             // first input:  matrix A
    const GxB_Scalar Thunk,         // optional input for the select operator
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

`GxB_Matrix_select` applies a select operator to the entries of a matrix, analogous to `T = A .* 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. If the operator is not type-generic, 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 $z = f(i, j, m, n, a_{ij}, \text{thunk})$, where `A` is m-by-n. 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`.

If `Thunk` is not `NULL`, it must be a valid `GxB_Scalar`. If it has no entry, it is treated as if it had a single entry equal to zero, for built-in types (not user-defined types).

For user-defined select operators, the entry is passed to the user-defined select operator, with no typecasting. Its type must be identical to `ttype` of the select operator.

For the `GxB_TRIL`, `GxB_TRIU`, `GxB_DIAG`, and `GxB_OFFDIAG`, the `Thunk` parameter may be `NULL`, or it may be present but contain no entry. In this case, these operators use the value of `k=0`, the main diagonal. If present, the `Thunk` can be any built-in type. The value of this entry is typecasted: `k = (int64_t) Thunk`. The value `k=0` specifies the main diagonal of the matrix, `k=1` is the +1 diagonal (the entries just above the main diagonal), `k=-1` is the -1 diagonal, and so on.

For the `GxB_*ZERO` select operators, `Thunk` is ignored, and may be `NULL`.

For built-in types, with the GxB_*THUNK operators, the value of Thunk is typecasted to the same type as the A matrix. For user-defined types, Thunk is passed to the select operator without typecasting.

The action of GxB_select with the built-in select 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 GxB_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 GxB_select, where T is a matrix.

The following operators may be used on matrices with a user-defined type: GxB_TRIL, GxB_TRIU, GxB_DIAG, GxB_OFFIAG, GxB_NONZERO, GxB_EQ_ZERO, GxB_NE_THUNK, and GxB_EQ_THUNK.

The comparators GxB_GT_* GxB_GE_* GxB_LT_*, and GxB_LE_* only work for built-in types. All other built-in select operators can be used for any type, both built-in and any user-defined type.

NOTE: For floating-point values, comparisons with NaN always return false. The built-in select operators should not be used with a scalar thunk 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	MATLAB analog	
GxB_TRIL	T=tril(A,k)	Entries in T are the entries on and below the kth diagonal of A.
GxB_TRIU	T=triu(A,k)	Entries in T are the entries on and above the kth diagonal of A.
GxB_DIAG	T=diag(A,k)	Entries in T are the entries on the kth diagonal of A.
GxB_OFFDIAG	T=A-diag(A,k)	Entries in T are all entries not on the kth diagonal of A.
GxB_NONZERO	T=A(A~=0)	Entries in T are all entries in A that have nonzero value.
GxB_EQ_ZERO	T=A(A==0)	Entries in T are all entries in A that are equal to zero.
GxB_GT_ZERO	T=A(A>0)	Entries in T are all entries in A that are greater than zero.
GxB_GE_ZERO	T=A(A<=0)	Entries in T are all entries in A that are greater than or equal to zero.
GxB_LT_ZERO	T=A(A<0)	Entries in T are all entries in A that are less than zero.
GxB_LE_ZERO	T=A(A<=0)	Entries in T are all entries in A that are less than or equal to zero.
GxB_NE_THUNK	T=A(A~=k)	Entries in T are all entries in A that are not equal to k.
GxB_EQ_THUNK	T=A(A==k)	Entries in T are all entries in A that are equal to k.
GxB_GT_THUNK	T=A(A>k)	Entries in T are all entries in A that are greater than k.
GxB_GE_THUNK	T=A(A>=k)	Entries in T are all entries in A that are greater than or equal to k.
GxB_LT_THUNK	T=A(A<k)	Entries in T are all entries in A that are less than k.
GxB_LE_THUNK	T=A(A<=k)	Entries in T are all entries in A that are less than or equal to k.

9.13 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.

9.13.1 GrB_Matrix_reduce_<op>: 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 <operator> reduce,        // reduce operator 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_<op>` is a generic name for two specific methods. Both methods reduce a matrix to a column vector using an operator, 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.

`GrB_Matrix_reduce_BinaryOp` relies on a binary operator for the reduction: the fourth argument `reduce`, a `GrB_BinaryOp`. All three domains of the operator must be the same. `GrB_Matrix_reduce_Monoid` performs the same reduction using a `GrB_Monoid` as its fourth argument. In both cases the reduction operator must be commutative and associative. Otherwise the results are undefined.

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.

9.13.2 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_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 reduction operator is a commutative and associative monoid with an identity value. Results are undefined if the monoid does not have these properties. This function differs from `GrB_Matrix_reduce_BinaryOp` (which reduces a matrix to a vector) in that it requires a valid monoid additive identity value. If the vector `u` has no entries, that identity value is copied into the scalar `t`. Otherwise, all of the entries in the vector are reduced to a single scalar using the `reduce` operator.

The scalar type is any of the built-in types, or a user-defined type. In the function signature it is a C type: `bool`, `int8_t`, ... `float`, `double`, or `void *` for a user-defined type. The user-defined 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.

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`.

Since this operation does not have a GraphBLAS input/output object, it cannot return an error string for `GrB_error`.

9.13.3 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_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 9.13.2.

Since this operation does not have a GraphBLAS input/output object, it cannot return an error string for `GrB_error`.

9.14 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. It may seem counter-intuitive, but this has the effect of not doing any transpose at all. As a result, `GrB_transpose` is useful for more than just transposing a matrix. It can be used as a direct interface to the accumulator/mask operation, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}$. This step also does any typecasting needed, so `GrB_transpose` can be used to typecast a matrix `A` into another matrix `C`. To do this, simply use `NULL` for the `Mask` and `accum`, and provide a non-default descriptor `desc` that sets the transpose option:

```
// C = typecasted copy of A
GrB_Descriptor_set (desc, GrB_INP0, GrB_TRAN) ;
GrB_transpose (C, NULL, NULL, A, desc) ;
```

If the types of `C` and `A` match, then the above two lines of code are the same as `GrB_Matrix_dup (&C, A)`, except that for `GrB_transpose` the matrix `C` must already exist and be the right size. If `C` does not exist, the work of `GrB_Matrix_dup` can be replicated with this:

```
// C = create an exact copy of A, just like GrB_Matrix_dup
GrB_Matrix C ;
GrB_Type type ;
GrB_Index nrows, ncols ;
GrB_Descriptor desc ;
GxB_Matrix_type (&type, A) ;
GrB_Matrix_nrows (&nrows, A) ;
```

```
GrB_Matrix_ncols (&ncols, A) ;
GrB_Matrix_new (&C, type, nrows, ncols) ;
GrB_Descriptor_new (&desc) ;
GrB_Descriptor_set (desc, GrB_INP0, GrB_TRAN) ;
GrB_transpose (C, NULL, NULL, A, desc) ;
```

Since the input matrix `A` is transposed by the descriptor, SuiteSparse:GraphBLAS does the right thing and does not transpose the matrix at all. Since `T = A` is not typecasted, SuiteSparse:GraphBLAS can construct `T` internally in $O(1)$ time and using no memory at all. This makes `Grb_transpose` a fast and direct interface to the accumulator/mask function in GraphBLAS.

This example is of course overkill, since the work can all be done by a single call to the `GrB_Matrix_dup` function. However, the `GrB_Matrix_dup` function can only create `C` as an exact copy of `A`, whereas variants of the code above can do many more things with these two matrices. For example, the `type` in the example can be replaced with any other type, perhaps selected from another matrix or from an operator.

Consider the following code excerpt, which uses `GrB_transpose` to remove all diagonal entries from a square matrix. It first creates a diagonal `Mask`, which is complemented so that $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A}$ does not modify the diagonal of $\mathbf{C}$. The `REPLACE` ensures that `C` is cleared first, and then $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A}$ modifies all entries in $\mathbf{C}$ where the mask $\mathbf{M}$ is false. These correspond to all the off-diagonal entries. The descriptor ensures that $\mathbf{A}$ is not transposed at all. The `Mask` can have any pattern, of course, and wherever it is set true, the corresponding entries in `A` are deleted from the copy `C`.

```
// remove all diagonal entries from the matrix A
// Mask = speye (n) ;
GrB_Matrix_new (&Mask, GrB_BOOL, n, n) ;
for (int64_t i = 0 ; i < n ; i++)
{
    GrB_Matrix_setElement (Mask, (bool) true, i, i) ;
}
// C<~Mask> = A, clearing C first.  No transpose.
GrB_Descriptor_new (&desc) ;
GrB_Descriptor_set (desc, GrB_INP0, GrB_TRAN) ;
GrB_Descriptor_set (desc, GrB_MASK, GrB_COMP) ;
GrB_Descriptor_set (desc, GrB_OUTP, GrB_REPLACE) ;
GrB_transpose (A, Mask, NULL, A, desc) ;
```

9.15 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, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \text{kron}(\mathbf{A}, \mathbf{B})$ where

$$\text{kron}(\mathbf{A}, \mathbf{B}) = \begin{bmatrix} a_{00} \otimes \mathbf{B} & \dots & a_{0,n-1} \otimes \mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} \otimes \mathbf{B} & \dots & a_{m-1,n-1} \otimes \mathbf{B} \end{bmatrix}$$

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.

10 Printing GraphBLAS objects

SPEC: The GraphBLAS API has no mechanism for printing the contents of GraphBLAS objects. This entire section is an extension to the specification.

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_SelectOp_fprint`	print and check a `GxB_SelectOp`
`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 `GxB_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`, nothing is printed (`pr` is effectively `GxB_SILENT`). 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; see the ANSI C `errno` or `GrB_error( )` for details.

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.

10.1 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)`.

10.2 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)`.

10.3 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`.

10.4 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`.

10.5 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`.

10.6 GxB_SelectOp_fprint: Print a GxB_SelectOp

```
GrB_Info GxB_SelectOp_fprint        // print and check a GxB_SelectOp
(
    GxB_SelectOp selectop,          // 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_SelectOp_fprint (GxB_TRIL, "tril", GxB_COMPLETE, f)` prints the `GxB_TRIL` select operator to the file `f`.

10.7 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`.

10.8 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`.

10.9 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`.

10.10 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`.

10.11 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`.

10.12 GxB_Scalar_fprint: Print a GxB_Scalar

```
GrB_Info GxB_Scalar_fprint          // print and check a GrB_Scalar
(
    GxB_Sclarr 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 sparse scalar `s` to the file `f`.

10.13 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.

11 Examples

Several examples of how to use GraphBLAS are listed below. They all appear in the `Demo` folder of SuiteSparse:GraphBLAS.

1. performing a breadth-first search,
2. finding a maximal independent set,
3. creating a random matrix,
4. creating a finite-element matrix,
5. reading a matrix from a file, and
6. complex numbers as a user-defined type.
7. triangle counting
8. PageRank
9. matrix import/export

Additional examples appear in the newly created LAGraph project, currently in progress.

11.1 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). As of Oct, 2019, the library includes the algorithms and utilities listed in the table below. Many additional algorithms are planned, such as betweenness centrality, PageRank, single-source shortest path (via delta stepping), minimum spanning trees, connected components, and many more. Refer to https://github.com/GraphBLAS/LAGraph for a current list of algorithms (the one below will soon be out of date). Most of the functions in the `Demo/` folder in SuiteSparse:GraphBLAS will eventually be translated into algorithms or utilities for LAGraph.

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.

Build `GraphBLAS` first, then the `LAGraph` library, and then the tests in `LAGraph/Test`.

Many of these algorithms are described in [ACD+20].

Algorithms	description
`LAGraph_bfs_pushpull`	a direction-optimized BFS [BAP12, YBO18], typically 2x faster than `bfs5m`
`LAGraph_bfs_simple`	a simple BFS (about the same as `bfs5m`)
`LAGraph_bc_batch`	batched betweenness-centrality
`LAGraph_bc`	betweenness-centrality
`LAGraph_cdlp`	community detection via label propagation
`LAGraph_cc`	connected components
`LAGraph_BF_*`	three variants of Bellman-Ford
`LAGraph_allktruss`	construct all k-trusses
`LAGraph_dnn`	sparse deep neural network [DAK19]
`LAGraph_ktruss`	construct a k-trusses
`LAGraph_lcc`	local clustering coefficient
`LAGraph_pagerank`	PageRank
`LAGraph_pagerank2`	PageRank variant
`LAGraph_tricount`	triangle counting

Utilities	description
`LAGraph_Vector_isall`	tests 2 vectors with a binary operator
`LAGraph_Vector_isequal`	tests if 2 vectors are equal
`LAGraph_Vector_to_dense`	converts a vector to dense
`LAGraph_alloc_global`	types, operators, monoids, and semirings
`LAGraph_finalize`	ends LAGraph
`LAGraph_free`	wrapper for `free`
`LAGraph_free_global`	frees objects created by `_alloc_global`
`LAGraph_get_nthreads`	get # of threads used
`LAGraph_grread`	read a binary matrix in Galois format
`LAGraph_init`	starts LAGraph
`LAGraph_isall`	tests 2 matrices with a binary operator
`LAGraph_isequal`	tests if 2 matrices are equal
`LAGraph_ispattern`	tests if all entries in a matrix are 1
`LAGraph_malloc`	wrapper for `malloc`
`LAGraph_mmread`	read a Matrix Market file
`LAGraph_mmwrite`	write a Matrix Market file
`LAGraph_pattern`	extracts the pattern of a matrix
`LAGraph_prune_diag`	diagonal entries from a matrix
`LAGraph_rand`	simple random number generator
`LAGraph_rand64`	`int64_t` random number generator
`LAGraph_random`	random matrix generator
`LAGraph_randx`	`double` random number generator
`LAGraph_set_nthreads`	set # of threads to use
`LAGraph_tic`	start a timer
`LAGraph_toc`	end a timer
`LAGraph_tsvread`	read a TSV file
`LAGraph_xinit`	starts LAGraph, with different malloc
`LAgraph_1_to_n`	construct the vector `1:n`
`GB_*sort*`	sorting for `LAGraph_cdlp`

11.2 Breadth-first search

The `bfs` examples in the `Demo` folder provide several examples of how to compute a breadth-first search (BFS) in GraphBLAS. Additional BFS examples are in LAGraph, shown below. The `LAGraph_bfs_simple` function starts at a given source node `s` of an undirected graph with `n` nodes. The graph is represented as an `n`-by-`n` matrix, `A`, where `A(i,j)` is the edge (i, j). The matrix `A` can have any type (even a user-defined type), since the `PAIR` operator does not access its values. No typecasting will be done.

The vector `v` of size `n` holds the level of each node in the BFS, where `v(i)=0` if the node has not yet been seen. This particular value makes `v` useful for another role. It can be used as a Boolean mask, since `0` is `false` and nonzero is `true`. Initially the entire `v` vector is zero. It is initialized as a dense vector, with all entries present, to improve performance (otherwise, it will slowly grow, incrementally, and this will take a lot of time if the number of BFS levels is high).

The vector `q` is the set of nodes just discovered at the current level, where `q(i)=true` if node `i` is in the current level. It starts out with just a single entry set to true, `q(s)`, the starting node.

Each iteration of the BFS consists of three calls to GraphBLAS. The first one uses `q` as a mask. It modifies all positions in `v` where `q` has an entry, setting them all to the current `level`.

```
// v<q> = level, using vector assign with q as the mask
GrB_assign (v, q, NULL, level, GrB_ALL, n, GrB_DESC_S) ;
```

The next call to GraphBLAS is the heart of the algorithm:

```
// q<!v> = q ||.&& A ; finds all the unvisited
// successors from current q, using !v as the mask
GrB_vxm (q, v, NULL, GxB_ANY_PAIR_BOOL, q, A, GrB_DESC_RC) ;
```

The vector `q` is all the set of nodes at the current level. Suppose `q(j)` is true, and it has a neighbor `i`. Then `A(i,j)=1`, and the dot product of `A(i,:)*q` using the `ANY_PAIR` semiring will use the `PAIR` multiplier on these two terms, `f (A(i,j), q(j))`, resulting in a value `1`. The `ANY` monoid will "sum" up all the results in this single row `i`; note that the `OR` monoid would compute the same thing. If the result is a column vector `t=A*q`, then this `t(i)` will be true. The vector `t` will be true for any node adjacent to any node in the set `q`.

Some of these neighbors of the nodes in `q` have already been visited by the BFS, either in the current level or in a prior level. These results must be discarded; what is desired is the set of all nodes `i` for which `t(i)` is true, and yet `v(i)` is still zero.

Enter the mask. The vector `v` is complemented for use a mask, via the `desc` descriptor. This means that wherever the vector is true, that position in the result is protected and will not be modified by the assignment. Only where `v` is false will the result be modified. This is exactly the desired result, since these represent newly seen nodes for the next level of the BFS. A node `k` already visited will have a nonzero `v(k)`, and thus `q(k)` will not be modified by the assignment.

The result `t` is written back into the vector `q`, through the mask, but to do this correctly, another descriptor parameter is used: `GrB_REPLACE`. The vector `q` was used to compute `t=A*q`, and after using it to compute `t`, the entire `q` vector needs to be cleared. Only new nodes are desired, for the next level. This is exactly what the `REPLACE` option does.

As a result, the vector `q` now contains the set of nodes at the new level of the BFS. It contains all those nodes (and only those nodes) that are neighbors of the prior set and that have not already been seen in any prior level.

A single call to `GrB_Vector_nvals` finds how many entries are in the current level. If this is zero, the BFS can terminate.

```
#include "LAGraph_internal.h"
#define LAGRAPH_FREE_ALL { GrB_free (&v) ; GrB_free (&q) ; }

GrB_Info LAGraph_bfs_simple     // push-only BFS
(
    GrB_Vector *v_output,   // v(i) is the BFS level of node i in the graph
    GrB_Matrix A,           // input graph, treated as if boolean in semiring
    GrB_Index source        // starting node of the BFS
)
{
    GrB_Info info ;
    GrB_Vector q = NULL ;           // nodes visited at each level
    GrB_Vector v = NULL ;           // result vector
    if (v_output == NULL) LAGRAPH_ERROR ("argument missing", GrB_NULL_POINTER) ;
    GrB_Index n, nvals ;
    GrB_Matrix_nrows (&n, A) ;
    // create an empty vector v, and make it dense
    GrB_Vector_new (&v, (n > INT32_MAX) ? GrB_INT64 : GrB_INT32, n) ;
    GrB_assign (v, NULL, NULL, 0, GrB_ALL, n, NULL) ;
    // create a boolean vector q, and set q(source) to true
    GrB_Vector_new (&q, GrB_BOOL, n) ;
    GrB_Vector_setElement (q, true, source) ;
    // BFS traversal and label the nodes
    for (int64_t level = 1 ; level <= n ; level++)
    {
        // v<q> = level
        GrB_assign (v, q, NULL, level, GrB_ALL, n, GrB_DESC_S) ;
        // break if q is empty
        GrB_Vector_nvals (&nvals, q) ;
        if (nvals == 0) break ;
        // q'<!v> = q'*A
        GrB_vxm (q, v, NULL, GxB_ANY_PAIR_BOOL, q, A, GrB_DESC_RC) ;
    }
    // free workspace and return result
    (*v_output) = v ;       // return result
    v = NULL ;              // set to NULL so LAGRAPH_FREE_ALL doesn't free it
    LAGRAPH_FREE_ALL ;      // free all workspace (except for result v)
    return (GrB_SUCCESS) ;
}
```

11.3 Maximal independent set

The *maximal independent set* problem is to find a set of nodes S such that no two nodes in S are adjacent to each other (an independent set), and all nodes not in S are adjacent to at least one node in S (and thus S is maximal since it cannot be augmented by any node while remaining an independent set). The `mis` function in the `Demo` folder solves this problem using Luby's method [Lub86]. The key operations in the method are replicated on the next page.

The gist of the algorithm is this. In each phase, all candidate nodes are given a random score. If a node has a score higher than all its neighbors, then it is added to the independent set. All new nodes added to the set cause their neighbors to be removed from the set of candidates. The process must be repeated for multiple phases until no new nodes can be added. This is because in one phase, a node `i` might not be added because one of its neighbors `j` has a higher score, yet that neighbor `j` might not be added because one of its neighbors `k` is added to the independent set instead. The node `j` is no longer a candidate and can never be added to the independent set, but node `i` could be added to S in a subsequent phase.

The initialization step, before the `while` loop, computes the degree of each node with a `PLUS` reduction. The set of `candidates` is Boolean vector, the `i`th component is true if node `i` is a candidate. A node with no neighbors causes the algorithm to stall, so these nodes are not candidates. Instead, they are immediately added to the independent set, represented by another Boolean vector `iset`. Both steps are done with an `assign`, using the `degree` as a mask, except the assignment to `iset` uses the complement of the mask, via the `sr_desc` descriptor. Finally, the `GrB_Vector_nvals` statement counts how many candidates remain.

Each phase of Luby's algorithm consists of 11 calls to GraphBLAS operations, all of which are either parallel, or take $O(1)$ time. Not all of them are described here since they are commented in the code itself. The two matrix-vector multiplications are the important parts and also take the most time. They also make interesting use of semirings and masks. The first one computes the largest score of all the neighbors of each node in the candidate set:

```
// compute the max probability of all neighbors
GrB_vxm (neighbor_max, candidates, NULL, maxFirst, prob, A, r_desc) ;
```

```
    // compute the degree of each node
    GrB_reduce (degrees, NULL, NULL, GrB_PLUS_FP64, A, NULL) ;

    // singletons are not candidates; they are added to iset first instead
    // candidates[degree != 0] = 1
    GrB_assign (candidates, degrees, NULL, true, GrB_ALL, n, NULL);

    // add all singletons to iset
    // iset[degree == 0] = 1
    GrB_assign (iset, degrees, NULL, true, GrB_ALL, n, sr_desc) ;

    // Iterate while there are candidates to check.
    GrB_Index nvals ;
    GrB_Vector_nvals (&nvals, candidates) ;

    while (nvals > 0)
    {
        // sparsify the random number seeds (just keep it for each candidate)
        GrB_assign (Seed, candidates, NULL, Seed, GrB_ALL, n, r_desc) ;
        // compute a random probability scaled by inverse of degree
        prand_xget (X, Seed) ;  // two calls to GrB_apply
        GrB_eWiseMult (prob, candidates, NULL, set_random, degrees, X, r_desc) ;
        // compute the max probability of all neighbors
        GrB_vxm (neighbor_max, candidates, NULL, maxFirst, prob, A, r_desc) ;
        // select node if its probability is > than all its active neighbors
        GrB_eWiseAdd (new_members, NULL,NULL, GrB_GT_FP64, prob, neighbor_max,0);
        // add new members to independent set.
        GrB_eWiseAdd (iset, NULL, NULL, GrB_LOR, iset, new_members, NULL) ;
        // remove new members from set of candidates c = c & !new
        GrB_apply (candidates, new_members, NULL, GrB_IDENTITY_BOOL,
            candidates, sr_desc) ;
        GrB_Vector_nvals (&nvals, candidates) ;
        if (nvals == 0) { break ; }                  // early exit condition
        // Neighbors of new members can also be removed from candidates
        GrB_vxm (new_neighbors, candidates, NULL, Boolean,
            new_members, A, NULL) ;
        GrB_apply (candidates, new_neighbors, NULL, GrB_IDENTITY_BOOL,
            candidates, sr_desc) ;
        GrB_Vector_nvals (&nvals, candidates) ;
    }
```

`A` is a symmetric Boolean matrix and `prob` is a sparse real vector (of type `FP32`), where `prob(i)` is nonzero only if node `i` is a candidate. The `prob` vector is computed from a random vector computed by a utility function `prand_xget`, in the `Demo` folder. It uses two calls to `GrB_apply` to construct

`n` random numbers in parallel, using a repeatable pseudo-random number generator.

The `maxFirst` semiring uses `z=FIRST(x,y)` as the multiplier operator. The column `A(:,j)` is the adjacency of node `j`, and the dot product `prob'*A(:,j)` applies the `FIRST` operator on all entries that appear in the intersection of `prob` and `A(:,j)`, where `z=FIRST(prob(i),A(i,j))` which is just `prob(i)` if `A(i,j)` is present. If `A(i,j)` not an explicit entry in the matrix, then this term is not computed and does not take part in the reduction by the `MAX` monoid.

Thus, each term `z=FIRST(prob(i),A(i,j))` is the score, `prob(i)`, of all neighbors `i` of node `j` that have a score. Node `i` does not have a score if it is not also a candidate and so this is skipped. These terms are then "summed" up by taking the maximum score, using `MAX` as the additive monoid.

Finally, the results of this matrix-vector multiply are written to the result, `neighbor_max`. The `r_desc` descriptor has the `REPLACE` option enabled. Since `neighbor_max` does not also take part in the computation `prob'*A`, it is simply cleared first. Next, is it modified only in those positions `i` where `candidates(i)` is true, using `candidates` as a mask. This sets the `neighbor_max` only for candidate nodes, and leaves the other components of `neighbor_max` as zero (implicit values not in the pattern of the vector).

All of the above work is done in a single matrix-vector multiply, with an elegant use of the `maxFirst` semiring coupled with a mask. The matrix-vector multiplication is described above as if it uses dot products of rows of `A` with the column vector `prob`, but SuiteSparse:GraphBLAS does not compute it that way. Sparse dot products are much slower the optimal method for multiplying a sparse matrix times a sparse vector. The result is the same, however.

The second matrix-vector multiplication is more straight-forward. Once the set of new members in the independent is found, it is used to remove all neighbors of those new members from the set of candidates.

The resulting method is very efficient. For the `Freescale2` matrix, the algorithm finds an independent set of size 1.6 million in 1.7 seconds (on the same MacBook Pro referred to in Section 11.2, using a single core), taking four iterations of the `while` loop. For comparison, removing its diagonal entries (required for the algorithm to work) takes 0.3 seconds in GraphBLAS (see Section 9.14), and simply transposing the matrix takes 0.24 seconds in both MATLAB and GraphBLAS.

11.4 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.

11.5 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. This example is taken from Loren Shure's blog on MATLAB Central, *Loren on the Art of MATLAB* [Dav07], which discusses the built-in `wathen` function. In MATLAB, 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. The difference in time between using `sparse` or `GrB_Matrix_build`, and using submatrix assignment with blocking mode (or in MATLAB which does not have a nonblocking mode) can be extreme. For the example matrix discussed in [Dav07], using `sparse` instead of submatrix assignment in MATLAB cut the run time of `wathen` from 305 seconds down to 1.6 seconds.

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.

11.6 Reading a matrix from a file

NOTE: 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. A comparison of `build` versus `setElement` has already been discussed in Section 11.4.

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 Speci-

fication). 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.

11.7 PageRank

The `Demo` folder contains three methods for computing the PageRank of the nodes of a graph. One uses floating-point arithmetic (`GrB_FP64`) and two user-defined unary operators (`dpagerank.c`). The second (`ipagerank.c`) is very similar, relying on integer arithmetic instead (`GrB_UINT64`). Neither method include a stopping condition. They simply compute a fixed number of iterations. The third example is more extensive (`dpagerank2.c`), and serves as an example of the power and flexibility of user-defined types, operators, monoids, and semirings. It creates a semiring for the entire PageRank computation. It terminates if the 2-norm of the change in the rank vector `r` is below a threshold.

11.8 Triangle counting

A triangle in an undirected graph is a clique of size three: three nodes i, j, and k that are all pairwise connected. There are many ways of counting the number of triangles in a graph. Let `A` be a symmetric matrix with values 0 and 1, and no diagonal entries; this matrix is the adjacency matrix of the graph. Let `E` be the edge incidence matrix with exactly two 1's per column. A column of `E` with entries in rows `i` and `j` represents the edge (i, j) in the graph, `A(i,j)=1` where `i<j`. Let `L` and `U` be the strictly lower and upper triangular parts of `A`, respectively.

The methods are listed in the table below. Most of them use a form of masked matrix-matrix multiplication. The methods are implemented in MATLAB in the `tricount.m` file, and in GraphBLAS in the `tricount.c` file, both in the `GraphBLAS/Demo` folder. Refer to the comments in those two files for details and derivations on how these methods work.

When the matrix is stored by row, and a mask is present and not complemented, `GrB_INP1` is `GrB_TRAN`, and `GrB_INP0` is `GxB_DEFAULT`, the SuiteSparse:GraphBLAS implementation of `GrB_mxm` always uses a dot-product formulation. Thus, the $\mathbf{C}\langle\mathbf{L}\rangle = \mathbf{L}\mathbf{U}^{\mathsf{T}}$ method uses dot products. This provides a mechanism for the end-user to select a masked dot product matrix multiplication method in SuiteSparse:GraphBLAS, which is occasionally faster than the outer product method. The MATLAB form assumes the matrices are stored by column (the only option in MATLAB).

Each method is followed by a reduction to a scalar, via `GrB_reduce` in GraphBLAS or by `nnz` or `sum(sum(...))` in MATLAB.

method and citation	in MATLAB	in GraphBLAS
minitri [WBS15]	`nnz(A*E==2)/3`	$\mathbf{C} = \mathbf{AE}$, then `GrB_apply`
Burkhardt [Bur16]	`sum(sum((A^2).*A))/6`	$\mathbf{C}\langle\mathbf{A}\rangle = \mathbf{A}^2$
Cohen [ABG15, Coh09]	`sum(sum((L*U).*A))/2`	$\mathbf{C}\langle\mathbf{A}\rangle = \mathbf{LU}$
Sandia [WDB+17]	`sum(sum((U*U).*U))`	$\mathbf{C}\langle\mathbf{L}\rangle = \mathbf{LL}$ (outer product)
SandiaDot	`sum(sum((U'*L).*L))`	$\mathbf{C}\langle\mathbf{U}\rangle = \mathbf{LU}^{\mathsf{T}}$ (dot product)
Sandia2	`sum(sum((L*L).*L))`	$\mathbf{C}\langle\mathbf{U}\rangle = \mathbf{UU}$ (outer product)

In general, the Sandia methods are the fastest of the 6 methods when implemented in GraphBLAS. For full details on the triangle counting and k-truss algorithms, and performance results, see [Dav18], a copy of which appears in the `SuiteSparse/GraphBLAS/Doc` folder. The code appears in `Extras`. That paper uses an earlier version of SuiteSparse:GraphBLAS in which all matrices are stored by column.

11.9 User-defined types and operators

The `Demo` folder contains two working examples of user-defined types, first discussed in Section 5.1.1: `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	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.

11.10 User applications using OpenMP or POSIX pthreads

Two example demo programs are included that illustrate how a multi-threaded user application can use GraphBLAS: `openmp_demo` uses OpenMP for its user threads and `pthread_demo` uses POSIX pthreads.

The `openmp_demo` can be compiled without OpenMP, in which case it becomes single-threaded. GraphBLAS can be compiled with or without OpenMP. This gives 6 different combinations, all of which are thread-safe.

Regardless of the threading model in the user application, GraphBLAS is always thread-safe. The results from the `openmp_demo` and `pthread_demo` programs 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, and these results are identical for all 6 combinations.

GraphBLAS can also be combined with user appplications that rely on MPI, the Intel TBB threading library, Microsoft Windows threads, or any other threading library. In all cases, GraphBLAS will be thread safe.

12 Compiling and Installing SuiteSparse:GraphBLAS

12.1 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. 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. For the Intel `icc` compiler, version 18.0 or later is required. Version 2.8.12 or later of `cmake` is required; version 3.0.0 is preferred.

If you are using a pre-C11 ANSI C compiler, or 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.

NOTE: icc is generally an excellent compiler, but it will generate slower code than gcc for v3.2.0 and later. This is merely because of how the two compilers treat `#pragma omp atomic read` and `#pragma omp atomic write`. The use of gcc for SuiteSparse:GraphBLAS v3.2.0 and later is recommended. This difference in performance should be resolved in a future version.

To compile SuiteSparse:GraphBLAS and the demo programs, simply type `make` in the main GraphBLAS folder, which compiles the library. To use a non-default compiler:

```
make CC=icc CXX=icc JOBS=4
```

After compiling the library, you can run the demos by typing `./demo` in the Demo folder.

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-6 cmake ..
CC=xlc cmake ..
CC=icc cmake ..
```

You can also do the following in the top-level GraphBLAS folder instead:

```
CC=gcc make
CC=gcc-6 cmake
CC=xlc cmake
CC=icc cmake
```

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
```

12.2 On Microsoft Windows

SuiteSparse:GraphBLAS is now ported to Microsoft Visual Studio. However, that compiler is not ANSI C11 compliant and does not support OpenMP v4.0. As a result, GraphBLAS on Windows will have a few 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`.
- Another limitation is the lack of support for OpenMP tasking, used in the parallel sort inside GraphBLAS. With Microsoft Visual Studio, the sort is compiled to use just a single thread. The sort is used for `GrB_Matrix_build` and `GrB_Vector_build`, and for `GrB_assign` and `GxB_subassign` when the index lists are unsorted on input. The internal sort still works as specified; it will just be single-threaded and thus these GraphBLAS functions will be slower on Windows as compared to Linux or MacOS.
- In addition, variable-length arrays are not supported, so user-defined types are limited to 128 bytes in size.

If you use a recent `gcc` or `icc` 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`.
   - 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 MATLAB interface discussed in the next section.

12.3 Compiling the MATLAB interface

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. Next:

1. In the MATLAB command window:

```
cd GraphBLAS/GraphBLAS/@GrB/private
gbmake
```

2. Follow the remaining instructions in the `GraphBLAS/GraphBLAS/README.md` file, to revise your MATLAB path and `startup.m` file.

3. As a quick test, try the MATLAB 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 12.2) Or, you might 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.

12.4 Default matrix format

By default, SuiteSparse:GraphBLAS stores its matrices by row, using the `GxB_BY_ROW` format. You can change the default at compile time to `GxB_BY_COL` using `cmake -DBYCOL=1`. For example:

```
cmake -DBYCOL=1 ..
```

The user application can also use `GxB_get` and `GxB_set` to set and query the global option (see also Sections 7.7 and 7.8):

```
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") ;
```

12.5 Setting the C flags and using CMake

The above options can also be combined. For example, to use the `gcc` compiler, to change the default format `GxB_FORMAT_DEFAULT` to `GxB_BY_COL`, use the following `cmake` command while in the `GraphBLAS/build` directory:

```
CC=gcc cmake -DBYCOL=1 ..
```

Then 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.

One particularly useful option is the `BURBLE` setting. It must be enabled both at compile time and then at run time with `GxB_set (GxB_BURBLE, true)`, or `GrB.burble(1)` in the MATLAB interface. If enabled, SuiteSparse:GraphBLAS will print out a report as to 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/Generated`, 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/Generated`. 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.

12.6 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`.

12.7 Running the Demos

By default, `make` in the top-level directory compiles the library and runs 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`.

12.8 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
```

12.9 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.

12.10 Cleaning up

To remove all compiled files, type `make distclean` in the top-level GraphBLAS folder.

13 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), and Carl Yang (UC Davis), 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. In particular, I would like to thank Tim Mattson for parallelizing the merge sort using OpenMP tasks. The parallel merge sort is used for `GrB_Matrix_build`, `GrB_Vector_build`, and some instances of `GrB_transpose`

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, and sparse matrices in MATLAB in particular.

I would like to thank Cleve Moler (MathWorks) for our many discussions on MATLAB, and for creating MATLAB in the first place. Without MATLAB, SuiteSparse:GraphBLAS would have been impossible to implement and test.

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 Roi Lipman, Redis Labs (https://redislabs.com), for our many discussions on GraphBLAS and 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/).

I would like to thank Lucas Jarman, MathWorks (http://mathworks.com), for his help in porting GraphBLAS to Microsoft Windows.

SuiteSparse:GraphBLAS was developed with support from NVIDIA, Intel, MIT Lincoln Lab, Redis Labs, IBM, and the National Science Foundation (1514406, 1835499).

14 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

[ABG15] A. Azad, A. Buluç, and J. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPSW 15*, pages 804–811. IEEE Computer Society, 2015.

[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. (accepted) https://www.siam.org/conferences/cm/conference/csc20.

[BAP12] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, 2012.

[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/.

[Bur16] P. Burkhardt. Graphing trillions of triangles. *Information Visualization*, 16:157–166, 2016. https://dx.doi.org/10.1177/1473871616666393.

[Coh09] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11(4):29–41, July 2009.

[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. The sparse data structure used in the book is much like the one in both MATLAB and SuiteSparse:GraphBLAS. 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

[Dav07] T. A Davis. Creating sparse finite-element matrices in MATLAB. *Loren on the Art of MATLAB*, Mar. 2007. Loren Shure, editor. Published by The MathWorks, Natick, MA. http://blogs.mathworks.com/loren/2007/03/01/creating-sparse-finite-element-matrices-in-matlab/.

[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] T. A. Davis. Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra. *ACM Trans. on Math. Software*, (to appear), 2019. http://faculty.cse.tamu.edu/davis/publications.html.

[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 worldfrom 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

[Lub86] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4), 1986. https://dx.doi.org/10.1137/0215074.

[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.

[WBS15] M. M. Wolf, J. W. Berry, and D. T. Stark. A task-based linear algebra building blocks approach for scalable graph analytics. In *IEEE HPEC'15*, pages 1–6. IEEE, 2015.

[WDB+17] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam. Fast linear algebra-based triangle counting with KokkosKernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2017. https://dx.doi.org/10.1109/HPEC.2017.8091043.

Triangle counting serves as a key building block for a set of important graph algorithms in network science. In this paper, we address the IEEE HPEC Static Graph Challenge problem of triangle counting, focusing on obtaining the best parallel

performance on a single multicore node. Our implementation uses a linear algebra-based approach to triangle counting that has grown out of work related to our miniTri data analytics miniapplication and our efforts to pose graph algorithms in the language of linear algebra. We leverage KokkosKernels to implement this approach efficiently on multicore architectures. Our performance results are competitive with the fastest known graph traversal-based approaches and are significantly faster than the Graph Challenge reference implementations, up to 670,000 times faster than the C++ reference and 10,000 times faster than the Python reference on a single Intel Haswell node.

[YBO18] Carl Yang, Aydın Buluç, and John D. Owens. Implementing push-pull efficiently in GraphBLAS. In *Proceedings of the International Conference on Parallel Processing*, ICPP 2018, pages 89:1–89:11, August 2018.