User Guide for SuiteSparse:GraphBLAS

Timothy A. Davis

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

http://suitesparse.com

https://people.engr.tamu.edu/davis

https://twitter.com/DocSparse

VERSION 10.0.0, Mar 1, 2025

Abstract

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

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

FIXME: add Doc updates for GraphBLAS v10:

```
NEW FOR v10:

    // GrB_GLOBAL, GrB_Matrix, GrB_Vector, GrB_Scalar: get/set
    GxB_ROWINDEX_INTEGER_HINT = 7053,   // hint for row indices
    GxB_COLINDEX_INTEGER_HINT = 7054,   // hint for column indices
    GxB_OFFSET_INTEGER_HINT = 7056,     // hint for offsets

    // GrB_Matrix, GrB_Vector, GrB_Scalar: get only
    GxB_ROWINDEX_INTEGER_BITS = 7057,   // # bits for row indices
    GxB_COLINDEX_INTEGER_BITS = 7058,   // # bits for column indices
    GxB_OFFSET_INTEGER_BITS = 7059,     // # bits for offsets
    GxB_IS_READONLY = 7078,

    GxB_OUTPUT_IS_READONLY = -7002,

    // GrB_get/GrB_set for GrB_Matrix:
    GxB_ISO = 7079,     // get: returns the current iso status
                        // set true: make the matrix iso-valued, if possible.
                        // set false: make the matrix non-iso-valued.

    GxB_INCLUDE_READONLY_STATISTICS

    // GrB_get/GrB_set for GrB_Descriptor:
    GxB_ROWINDEX_LIST = 7062,       // how GrB_Vector I is intrepretted
    GxB_COLINDEX_LIST = 7063,       // how GrB_Vector J is intrepretted
    GxB_VALUE_LIST = 7064,          // how GrB_Vector X is intrepretted

// settings for GxB_ROWINDEX_LIST, GxB_COLINDEX_LIST, and GxB_VALUE_LIST:
#define GxB_USE_VALUES (0)      /* use the values of the vector (default) */
#define GxB_USE_INDICES (7060)  /* use the indices of the vector */
#define GxB_IS_STRIDE (7061)    /* use the values, of size 3, for lo:hi:inc */

new Container methods:

    struct GxB_Container_struct

new _Vector methods for build, extract, extractTuples, assign, subassign:
```

```
GrB_assign polymorphic method extended
GxB_Row_assign_Vector
GxB_Col_assign_Vector
GxB_Matrix_assign_Scalar_Vector
GxB_Matrix_assign_Vector
GxB_Vector_assign_Scalar_Vector
GxB_Vector_assign_Vector

GxB_subassign polymorphic method extended
GxB_Row_subassign_Vector
GxB_Col_subassign_Vector
GxB_Matrix_subassign_Scalar_Vector
GxB_Matrix_subassign_Vector
GxB_Vector_subassign_Scalar_Vector
GxB_Vector_subassign_Vector

GxB_extract polymorphic method extended
GxB_Col_extract_Vector
GxB_Matrix_extract_Vector
GxB_Vector_extract_Vector

GxB_Matrix_build_Scalar_Vector
GxB_Matrix_build_Vector
GxB_Vector_build_Scalar_Vector
GxB_Vector_build_Vector

GxB_Matrix_extractTuples_Vector
GxB_Vector_extractTuples_Vector

GxB_Container_new
GxB_Container_free
GxB_load_Matrix_from_Container
GxB_load_Vector_from_Container
GxB_unload_Matrix_into_Container
GxB_unload_Vector_into_Container
GxB_Vector_load
GxB_Vector_unload

The Container->[phbix] GrB_Vectors should not be freed by the user
application, just loaded/unloaded with GxB_Vector load/unload.  All of them
```

must be present for the Container to be used, for any matrix format. They are allocated when the Container is created, and freed when the Container is freed. If they are somehow freed, simply free the entire Container and recreate it, or replace the missing vectors with length-0 vectors of any type; for example:

```
GrB_Vector_new (&(Container->p), GrB_UINT32, 0) ;
```

GrB_Field enum deprecated

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, Dav23, Dav18, DAK19, ACD+20, MDK+19].

A full and precise definition of the GraphBLAS specification is provided in *The GraphBLAS C API Specification* by Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira, Carl Yang, and Benjamin Brock [BMM+17a, BMM+17b, BBM+21], based on *GraphBLAS Mathematics* by Jeremy Kepner [Kep17]. The GraphBLAS C API Specification is available at http://graphblas.org. This version of SuiteSparse:GraphBLAS conforms to Version 2.1.0 (Dec 22, 2023) of *The GraphBLAS C API specification*.

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

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

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

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

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

2.1 Graphs and sparse matrices

Graphs can be huge, with many nodes and edges. A dense adjacency matrix $\mathbf{A}$ for a graph of n nodes takes $O(n^2)$ memory, which is impossible if n is, say, a million. Let $|\mathbf{A}|$ denote the number of entries in a matrix. Most graphs arising in practice are sparse, however, with only $|\mathbf{A}| = O(n)$ edges, where $|\mathbf{A}|$ denotes the number of edges in the graph, or the number of explicit entries present in the data structure for the matrix $\mathbf{A}$. Sparse graphs with millions of nodes and edges can easily be created by representing them as sparse matrices, where only explicit values need to be stored. Some graphs are *hypersparse*, with $|\mathbf{A}| << n$. SuiteSparse:GraphBLAS supports three kinds of sparse matrix formats: a regular sparse format, taking $O(n + |\mathbf{A}|)$ space, a hypersparse format taking only $O(|\mathbf{A}|)$ space, and a bitmap form, taking $O(n^2)$ space. Full matrices are also represented in $O(n^2)$ space. Using its hypersparse format, creating a sparse matrix of size n-by-n where $n = 2^{60}$ (about 10^{18}) can be done on quite easily on a commodity laptop, limited only by $|\mathbf{A}|$. To the GraphBLAS user application, all matrices look alike, since these formats are opaque, and SuiteSparse:GraphBLAS switches between them at will.

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

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

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

2.2 Overview of GraphBLAS methods and operations

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

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

GraphBLAS can do far more than what MATLAB/Octave 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 12. With this brief overview, the full scope of GraphBLAS extensions of these operations can now be described.

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

The scalar addition of conventional matrix multiplication is replaced with a *monoid*. A monoid is an associative and commutative binary operator `z=f(x,y)` where all three domains are the same (the types of `x`, `y`, and `z`), and where the operator has an identity value `id` such that `f(x,id)=f(id,x)=x`. Performing matrix multiplication with a semiring uses a monoid in place of the "add" operator, scalar addition being just one of many possible monoids. The identity value of addition is zero, since $x + 0 = 0 + x = x$. GraphBLAS includes many built-in operators suitable for use as a monoid: min (with an identity value of positive infinity), max (whose identity is negative infinity), add (identity is zero), multiply (with an identity of one), four logical operators: AND, OR, exclusive-OR, and Boolean equality (XNOR), four bitwise operators (AND, OR, XOR, and XNOR), and the ANY operator See Section 6.3.6 for more details on the unusual ANY operator. User-created monoids can be defined with any associative and commutative operator that has an identity value.

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

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

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

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

Consider element-wise multiplication, `C=A.*B` in MATLAB. The operator (multiply in this case) is applied to entries in the set intersection, and the pattern of `C` just this set intersection. Entries in `A` but not `B`, or visa-versa, do not appear in `C`. In GraphBLAS, any binary operator can be used in this manner, not just scalar multiplication. The difference between element-wise "add" and "multiply" is not the operators, but whether or not the pattern of the result is the set union or the set intersection. In both cases, the operator is only applied to the set intersection.

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

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

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

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

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

2.3 The accumulator and the mask

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

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

This process can be described in mathematical notation as:

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

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

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

if `accum` is `NULL`

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

else

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

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

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

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

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

$z_{ij} = c_{ij}$

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

$z_{ij} = t_{ij}$

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

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

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

if `Mask` is `NULL`

if (`GrB_COMP`)

$\mathbf{C}$ is not modified

else

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

else

if (`GrB_COMP`)

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

else

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4 Typecasting

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

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

However, unlike MATLAB, the C11 language specification states that the results of typecasting a `float` or `double` to an integer type is not always defined. In SuiteSparse:GraphBLAS, whenever C leaves the result undefined the rules used in MATLAB are followed. In particular `+Inf` converts to the largest integer value, `-Inf` converts to the smallest (zero for unsigned integers), and `NaN` converts to zero. Positive values outside the range of the integer are converted to the largest positive integer, and negative values less than the most negative integer are converted to that most negative integer. Other than these special cases, SuiteSparse:GraphBLAS trusts the C compiler for the rest of its typecasting.

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

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

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

2.5 Notation and list of GraphBLAS operations

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

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

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

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

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

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

The MATLAB/Octave interface to SuiteSparse:GraphBLAS is included with this distribution, described in Section 3.1. Python, Julia, Go, and Java interfaces are available. These are not part of the SuiteSparse:GraphBLAS distribution. See the links below.

3.1 MATLAB/Octave Interface

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

The MATLAB/Octave interface adds the `@GrB` class, which is an opaque MATLAB/Octave object that contains a GraphBLAS matrix, either double or single precision (real or complex), boolean, or any of the built-in integer types. MATLAB/Octave 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/Octave, or `help GrB` for more details.

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

- `GrB` matrices with dimension larger than `2^53` do not display properly in the `whos` command. The size is displayed correctly with `disp` or `display`.
- Non-blocking mode is not exploited.
- Linear indexing: `A(:)` for a 2D matrix, and `I=find(A)`.
- Singleton expansion.

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

3.2 Python Interface

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

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

3.3 Julia Interface

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

3.4 Go Interface

Pascal Costanza (Intel) has a Go interface to GraphBLAS and LAGraph:

- forGraphBLASGo: https://github.com/intel/forGraphBLASGo, which is almost a complete wrapper for SuiteSparse:GraphBLAS. Documentation is at https://pkg.go.dev/github.com/intel/forGraphBLASGo.

- forLAGraphGo: https://github.com/intel/forLAGraphGo, which is in progress. Documentation is at https://pkg.go.dev/github.com/intel/forLAGraphGo.

3.5 Java Interface

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

4 Performance of MATLAB versus GraphBLAS

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

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

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

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

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

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

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

5 GraphBLAS Initialization/Finalization

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1 GrB_Index: the GraphBLAS integer

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

```
typedef uint64_t GrB_Index ;
```

Row and column indices of an `nrows`-by-`ncols` matrix range from zero to the `nrows-1` for the rows, and zero to `ncols-1` for the columns. Indices are zero-based, like C, and not one-based, like MATLAB/Octave. In SuiteSparse:GraphBLAS, the largest permitted index value is `GrB_INDEX_MAX`, defined as $2^{60} - 1$. The largest permitted matrix or vector dimension is 2^{60} (that is, `GrB_INDEX_MAX+1`). The largest `GrB_Matrix` that SuiteSparse: GraphBLAS can construct is thus 2^{60}-by-2^{60}. An n-by-n matrix $\mathbf{A}$ that size can easily be constructed in practice with $O(|\mathbf{A}|)$ memory requirements, where $|\mathbf{A}|$ denotes the number of entries that explicitly appear in the pattern of $\mathbf{A}$. The time and memory required to construct a matrix that large does not depend on n, since SuiteSparse:GraphBLAS can represent $\mathbf{A}$ in hypersparse form (see Section 10.10.2). The largest `GrB_Vector` that can be constructed is 2^{60}-by-1.

5.2 GrB_init: initialize GraphBLAS

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

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

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

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

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

SuiteSparse:GraphBLAS is multithreaded internally, via OpenMP, and it is also safe to use in a multithreaded user application. See Section 18 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 inputs to GraphBLAS operations or methods, but only if they have no pending operations (use `GrB_wait` first). User threads cannot simultaneously modify a matrix or vector via any GraphBLAS operation or method.

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

With multiple user threads, exactly one user thread must call `GrB_init` before any user thread may call any `GrB_*` or `GxB_*` function. When the user application is finished, exactly one user thread must call `GrB_finalize`, after which no user thread may call any `GrB_*` or `GxB_*` function. The mode of a GraphBLAS session can be queried with `GrB_get`; see Section 10 for details.

5.3 GrB_getVersion: determine the C API Version

```
GrB_Info GrB_getVersion         // run-time 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 run-time access of the version of the C API Specification supported by the library.

5.4 GxB_init: initialize with alternate malloc

```
GrB_Info GxB_init           // start up GraphBLAS and also define malloc
(
    int mode,               // blocking or non-blocking mode (GrB_Mode)
    // pointers to memory management functions.
    void * (* user_malloc_func  ) (size_t),
    void * (* user_calloc_func  ) (size_t, size_t),
    void * (* user_realloc_func ) (void *, size_t),
    void   (* user_free_func    ) (void *)
) ;
```

`GxB_init` is identical to `GrB_init`, except that it also redefines the memory management functions that SuiteSparse:GraphBLAS will use. Giving the user application control over this is particularly important when using the `GxB_*serialize` and `GxB_Container` methods described in Section 6.11 and **??**, since they require the user application and GraphBLAS to use the same memory manager. `user_calloc_func` and `user_realloc_func` are optional, and may be `NULL`. If `NULL`, then the `user_malloc_func` is relied on instead, for all memory allocations. These functions can only be set once, when GraphBLAS starts. They can be queried using `GrB_get` (see Section 10.2). Either `GrB_init` or `GxB_init` must be called before any other GraphBLAS operation, but not both. The functions passed to `GxB_init` must be thread-safe. The following usage is identical to `GrB_init(mode)`:

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

5.5 GrB_Info: status code returned by GraphBLAS

Each GraphBLAS method and operation returns its status to the caller as its return value, an enumerated type (an `enum`) called `GrB_Info`. The first two values in the following table denote a successful status, the rest are error codes.

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

Error	value	description
`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.
`GxB_EXHAUSTED`	2	the iterator is exhausted
`GrB_UNINITIALIZED_OBJECT`	-1	object has not been initialized
`GrB_NULL_POINTER`	-2	input pointer is `NULL`
`GrB_INVALID_VALUE`	-3	generic error code; some value is bad
`GrB_INVALID_INDEX`	-4	a row or column index is out of bounds
`GrB_DOMAIN_MISMATCH`	-5	object domains are not compatible
`GrB_DIMENSION_MISMATCH`	-6	matrix dimensions do not match
`GrB_OUTPUT_NOT_EMPTY`	-7	output matrix already has values in it
`GrB_NOT_IMPLEMENTED`	-8	not implemented in SS:GrB
`GrB_ALREADY_SET`	-9	field already written to
`GrB_PANIC`	-101	unrecoverable error
`GrB_OUT_OF_MEMORY`	-102	out of memory
`GrB_INSUFFICIENT_SPACE`	-103	output array not large enough
`GrB_INVALID_OBJECT`	-104	object is corrupted
`GrB_INDEX_OUT_OF_BOUNDS`	-105	a row or column index is out of bounds
`GrB_EMPTY_OBJECT`	-106	a input scalar has no entry
`GxB_JIT_ERROR`	-1001	JIT compiler error

5.6 GrB_error: get more details on the last error

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

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

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

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

SuiteSparse:GraphBLAS reports many helpful details via `GrB_error`. For example, if a row or column index is out of bounds, the report will state what those bounds are. If a matrix dimension is incorrect, the mismatching dimensions will be provided. Refer to the output of the example programs in the `Demo` and `Test` folder, which intentionally generate errors to illustrate the use of `GrB_error`.

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

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

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

5.7 GrB_finalize: finish GraphBLAS

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

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

6 GraphBLAS Objects and their Methods

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

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

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

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

6.1 The GraphBLAS type: GrB_Type

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

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

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

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

GraphBLAS function	purpose	Section
`GrB_Type_new`	create a user-defined type	6.1.1
`GxB_Type_new`	create a user-defined type, with name and definition	6.1.2
`GrB_Type_wait`	wait for a user-defined type	6.1.3
`GrB_get`	get properties of a type	10.3
`GrB_set`	set the type name/definitition	10.3
`GxB_Type_from_name`	return the type from its name	6.1.4
`GrB_Type_free`	free a user-defined type	6.1.5

6.1.1 GrB_Type_new: create a user-defined type

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

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

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

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

```
typedef struct
{
    float stuff [4][4] ;
    char whatstuff [64] ;
}
wildtype ;
GrB_Type WildType ;
GrB_Type_new (&WildType, sizeof (wildtype)) ;
```

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

Performing arithmetic on matrices and vectors with user-defined types requires operators to be defined. Refer to Section 17.5 for more details on these example user-defined types.

User defined types created by `GrB_Type_new` will not work with the JIT; use `GxB_Type_new` instead.

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

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

`GxB_Type_new` creates a type with a name and definition that are known to GraphBLAS, as strings. The `type_name` is any valid string (max length of

128 characters, including the required null-terminating character) that may appear as the name of a C type created by a C `typedef` statement. It must not contain any white-space characters. For example, to create a type of size 16*4+1 = 65 bytes, with a 4-by-4 dense float array and a 32-bit integer:

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

The `type_name` and `type_defn` are both null-terminated strings. The two strings are optional, but are required to enable the JIT compilation of kernels that use this type. At most `GxB_MAX_NAME_LEN` characters are accessed in `type_name`; characters beyond that limit are silently ignored.

If the `sizeof_ctype` is zero, and the strings are valid, a JIT kernel is compiled just to determine the size of the type. This is feature useful for interfaces in languages other than C, which could create valid strings for C types but would not have a reliable way to determine the size of the type.

The above example is identical to the following usage, except that `GrB_Type_new` requires `sizeof_ctype` to be nonzero, and equal to the size of the C type.

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

6.1.3 GrB_Type_wait: wait for a type

```
GrB_Info GrB_wait               // wait for a user-defined type
(
    GrB_Type type,              // type to wait for
    int mode                    // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

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

6.1.4 GxB_Type_from_name: return the type from its name

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

Returns the built-in type from the corresponding name of the type. The following examples both return `type` as `GrB_BOOL`.

```
GxB_Type_from_name (&type, "bool") ;
GxB_Type_from_name (&type, "GrB_BOOL") ; }
```

If the name is from a user-defined type, the `type` is returned as `NULL`. This is not an error condition. The user application must itself do this translation since GraphBLAS does not keep a registry of all user-defined types.

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

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

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

// later on, to query the type of A:
size_t typesize ;
GrB_Scalar_new (s, GrB_UINT64) ;
GrB_get (type, s, GrB_SIZE) ;
GrB_Scalar_extractElement (&typesize, GrB_UINT64) ;
GrB_Type atype ;
char atype_name [GxB_MAX_NAME_LEN] ;
GrB_get (A, atype_name, GrB_EL_TYPE_STRING) ;
GxB_Type_from_name (&atype, atype_name) ;
if (atype == NULL)
{
    // This is not yet an error.  It means that A has a user-defined type.
    if ((strcmp (atype_name, "myfirsttype")) == 0) atype = MyType1 ;
```

```
    else if ((strcmp (atype_name, "myquaternion")) == 0) atype = MyQType ;
    else { ... this is now an error ... the type of A is unknown.  }
}
```

6.1.5 GrB_Type_free: free a user-defined type

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

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

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

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

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

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

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

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

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

In the notation in the tables below, T is any of the 13 built-in types and is a 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`), Z refers to any complex floating point type (`FC32` or `FC64`), and N refers to `INT32` or `INT64`.

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

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

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

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

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

Unary operators for floating-point types (real only)			
GraphBLAS name	types (domains)	$z = f(x)$	description
`GxB_LGAMMA_`R	$R \rightarrow R$	$z = \log(\lvert\Gamma(x)\rvert)$	log of gamma function
`GxB_TGAMMA_`R	$R \rightarrow R$	$z = \Gamma(x)$	gamma function
`GxB_ERF_`R	$R \rightarrow R$	$z = \mathrm{erf}(x)$	error function
`GxB_ERFC_`R	$R \rightarrow R$	$z = \mathrm{erfc}(x)$	complimentary error function
`GxB_CBRT_`R	$R \rightarrow R$	$z = x^{1/3}$	cube root
`GxB_FREXPX_`R	$R \rightarrow R$	$z = \mathrm{frexpx}(x)$	normalized fraction
`GxB_FREXPE_`R	$R \rightarrow R$	$z = \mathrm{frexpe}(x)$	normalized exponent

Unary operators for complex types			
GraphBLAS name	types (domains)	$z = f(x)$	description
GxB_CONJ_Z	$Z \rightarrow Z$	$z = \overline{x}$	complex conjugate
GxB_ABS_Z	$Z \rightarrow F$	$z = \lvert x \rvert$	absolute value
GxB_CREAL_Z	$Z \rightarrow F$	$z = \text{real}(x)$	real part
GxB_CIMAG_Z	$Z \rightarrow F$	$z = \text{imag}(x)$	imaginary part
GxB_CARG_Z	$Z \rightarrow F$	$z = \text{carg}(x)$	angle

Built-in index-based unary operators return the row or column index of an entry. For a matrix $z = f(a_{ij})$ returns $z = i$ or $z = j$, or $+1$ for 1-based indices. The latter is useful in the MATLAB/Octave interface, where row and column indices are 1-based. When applied to a vector, j is always zero, and i is the index in the vector. These built-in unary operators come in two types: INT32 and INT64, which is the type of the output, z. The functions are agnostic to the type of their inputs; they only depend on the position of the entries, not their values. User-defined index-based operators cannot be defined by GrB_UnaryOp_new; use GrB_IndexUnaryOp_new instead; see Section 6.4.

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

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

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

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

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

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

GraphBLAS function	purpose	Section
`GrB_UnaryOp_new`	create a user-defined unary operator	6.2.1
`GxB_UnaryOp_new`	create a named user-defined unary operator	6.2.2
`GrB_UnaryOp_wait`	wait for a user-defined unary operator	6.2.3
`GrB_get`	get properties of an operator	10.4
`GrB_set`	set the operator name/definition	10.4
`GrB_UnaryOp_free`	free a user-defined unary operator	6.2.4

6.2.1 GrB_UnaryOp_new: create a user-defined unary operator

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

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

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

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

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

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

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

6.2.2 GxB_UnaryOp_new: create a named user-defined unary operator

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

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

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

The two strings unop_name and unop_defn are optional, but are required to enable the JIT compilation of kernels that use this operator.

If JIT compilation is enabled, or if the corresponding JIT kernel has been copied into the PreJIT folder, the function may be NULL. In this case, a JIT kernel is compiled that contains just the user-defined function. If the JIT is disabled and the function is NULL, this method returns GrB_NULL_POINTER.

The above example is identical to the following usage, except that GrB_UnaryOp_new requires a non-NULL function pointer.

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

6.2.3 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
    int mode                    // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

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

6.2.4 GrB_UnaryOp_free: free a user-defined unary operator

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

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

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

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

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

A binary operator is a scalar function of the form $z = f(x, y)$. The types of z, x, and y need not be the same. The built-in binary operators are listed in the tables below. The notation T refers to any of the 13 built-in types, but two of those types are SuiteSparse extensions (`GxB_FC32` and `GxB_FC64`). For those types, the operator name always starts with `GxB`, not `GrB`). The notation R refers to any real type (all but `FC32` and `FC64`).

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

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

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

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

Binary operators for all non-complex types			
GraphBLAS name	types (domains)	$z = f(x, y)$	description
GrB_MIN_R	$R \times R \rightarrow R$	$z = \min(x, y)$	minimum
GrB_MAX_R	$R \times R \rightarrow R$	$z = \max(x, y)$	maximum
GxB_ISGT_R	$R \times R \rightarrow R$	$z = (x > y)$	greater than
GxB_ISLT_R	$R \times R \rightarrow R$	$z = (x < y)$	less than
GxB_ISGE_R	$R \times R \rightarrow R$	$z = (x \geq y)$	greater than or equal
GxB_ISLE_R	$R \times R \rightarrow R$	$z = (x \leq y)$	less than or equal
GxB_LOR_R	$R \times R \rightarrow R$	$z = (x \neq 0) \vee (y \neq 0)$	logical OR
GxB_LAND_R	$R \times R \rightarrow R$	$z = (x \neq 0) \wedge (y \neq 0)$	logical AND
GxB_LXOR_R	$R \times R \rightarrow R$	$z = (x \neq 0) \veebar (y \neq 0)$	logical XOR

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

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

Binary comparators for non-complex types			
GraphBLAS name	types (domains)	$z = f(x, y)$	description
GrB_GT_R	$R \times R \rightarrow$`bool`	$z = (x > y)$	greater than
GrB_LT_R	$R \times R \rightarrow$`bool`	$z = (x < y)$	less than
GrB_GE_R	$R \times R \rightarrow$`bool`	$z = (x \geq y)$	greater than or equal
GrB_LE_R	$R \times R \rightarrow$`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` $\rightarrow$ `bool`	$z = x \vee y$	logical OR
GrB_LAND	`bool` $\times$ `bool` $\rightarrow$ `bool`	$z = x \wedge y$	logical AND
GrB_LXOR	`bool` $\times$ `bool` $\rightarrow$ `bool`	$z = x \veebar y$	logical XOR
GrB_LXNOR	`bool` $\times$ `bool` $\rightarrow$ `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 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	$F \times F \rightarrow F$	$z = \tan^{-1}(y/x)$	4-quadrant arc tangent
GxB_HYPOT_F	$F \times F \rightarrow F$	$z = \sqrt{x^2 + y^2}$	hypotenuse
GxB_FMOD_F	$F \times F \rightarrow F$		C11 fmod
GxB_REMAINDER_F	$F \times F \rightarrow F$		C11 remainder
GxB_LDEXP_F	$F \times F \rightarrow F$		C11 ldexp
GxB_COPYSIGN_F	$F \times F \rightarrow F$		C11 copysign
GxB_CMPLX_F	$F \times F \rightarrow Z$	$z = x + y \times i$	complex from real & imag

Eight bitwise operators are predefined for signed and unsigned integers.

Binary operators for signed and unsigned integers			
GraphBLAS name	types (domains)	$z = f(x,y)$	description
GrB_BOR_I	$I \times I \rightarrow I$	z=x|y	bitwise logical OR
GrB_BAND_I	$I \times I \rightarrow I$	z=x&y	bitwise logical AND
GrB_BXOR_I	$I \times I \rightarrow I$	z=x^y	bitwise logical XOR
GrB_BXNOR_I	$I \times I \rightarrow I$	z=~(x^y)	bitwise logical XNOR
GxB_BGET_I	$I \times I \rightarrow I$		get bit y of x
GxB_BSET_I	$I \times I \rightarrow I$		set bit y of x
GxB_BCLR_I	$I \times I \rightarrow I$		clear bit y of x
GxB_BSHIFT_I	$I\times$int8$\rightarrow I$		bit shift

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

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

Floating-point operations follow the IEEE 754 standard. Thus, computing $x/0$ for a floating-point x results in +Inf if x is positive, -Inf if x is negative, and NaN if x is zero. The application is not terminated. However, integer division by zero normally terminates an application. SuiteSparse:GraphBLAS avoids this by adopting the same rules as MATLAB, which are analogous to how the IEEE standard handles floating-point division by zero. For integers, when x is positive, $x/0$ is the largest positive integer, for negative x it is the minimum integer, and 0/0 results in zero. For example, for an integer x of type GrB_INT32, 1/0 is $2^{31}-1$ and (-1)/0 is -2^{31}. Refer to Section 6.1 for a list of integer ranges.

6.3.1 GraphBLAS binary operators based on index binary operators

Eight binary operators based on underlying index binary operators are predefined. They differ when used in a semiring and when used in GrB_eWise* and GrB_apply. These index-based binary operators cannot be used in GrB_build, nor can they be used as the accum operator for any operation.

The index-based binary operators do not depend on the type or numerical value of their inputs, just their position in a matrix or vector. For a vector, j is always 0, and i is the index into the vector. There are two types N available: INT32 and INT64, which is the type of the output z. User-defined index-based operators are not defined by GrB_BinaryOp_new, but by GxB_BinaryOp_new_IndexOp instead. See Section 6.5 for details.

Index-based binary operators for any type (including user-defined) when used as a multiplicative operator in a semiring			
GraphBLAS name	types (domains)	$z = f(a_{ik}, b_{kj})$	description
GxB_FIRSTI_N	$\rightarrow N$	$z = i$	row index of a_{ik} (0-based)
GxB_FIRSTI1_N	$\rightarrow N$	$z = i + 1$	row index of a_{ik} (1-based)
GxB_FIRSTJ_N	$\rightarrow N$	$z = k$	column index of a_{ik} (0-based)
GxB_FIRSTJ1_N	$\rightarrow N$	$z = k + 1$	column index of a_{ik} (1-based)
GxB_SECONDI_N	$\rightarrow N$	$z = k$	row index of b_{kj} (0-based)
GxB_SECONDI1_N	$\rightarrow N$	$z = k + 1$	row index of b_{kj} (1-based)
GxB_SECONDJ_N	$\rightarrow N$	$z = j$	column index of b_{kj} (0-based)
GxB_SECONDJ1_N	$\rightarrow N$	$z = j + 1$	column index of b_{kj} (1-based)

Index-based binary operators for any type (including user-defined) when used in all other methods			
GraphBLAS name	types (domains)	$z = f(a_{ij}, b_{ij})$	description
GxB_FIRSTI_N	$\rightarrow N$	$z = i$	row index of a_{ij} (0-based)
GxB_FIRSTI1_N	$\rightarrow N$	$z = i + 1$	row index of a_{ij} (1-based)
GxB_FIRSTJ_N	$\rightarrow N$	$z = j$	column index of a_{ij} (0-based)
GxB_FIRSTJ1_N	$\rightarrow N$	$z = j + 1$	column index of a_{ij} (1-based)
GxB_SECONDI_N	$\rightarrow N$	$z = i$	row index of b_{ij} (0-based)
GxB_SECONDI1_N	$\rightarrow N$	$z = i + 1$	row index of b_{ij} (1-based)
GxB_SECONDJ_N	$\rightarrow N$	$z = j$	column index of b_{ij} (0-based)
GxB_SECONDJ1_N	$\rightarrow N$	$z = j + 1$	column index of b_{ij} (1-based)

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

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

GraphBLAS function	purpose	Section
`GrB_BinaryOp_new`	create a user-defined binary operator	6.3.2
`GxB_BinaryOp_new`	create a named user-defined binary operator	6.3.3
`GrB_BinaryOp_wait`	wait for a user-defined binary operator	6.3.4
`GrB_get`	get properties of an operator	10.6
`GrB_set`	set the operator name/definition	10.6
`GrB_BinaryOp_free`	free a user-defined binary operator	6.3.5

6.3.2 GrB_BinaryOp_new: create a user-defined binary operator

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

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

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

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

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

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

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

6.3.3 GxB_BinaryOp_new: create a named user-defined binary operator

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

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

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

The two strings `binop_name` and `binop_defn` are optional, but are required to enable the JIT compilation of kernels that use this operator.

If the JIT is enabled, or if the corresponding JIT kernel has been copied into the `PreJIT` folder, the `function` may be `NULL`. In this case, a JIT kernel is compiled that contains just the user-defined function. If the JIT is disabled and the `function` is `NULL`, this method returns `GrB_NULL_POINTER`.

The above example is identical to the following usage, except that `GrB_BinaryOp_new` requires a non-NULL function pointer.

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

6.3.4 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
    int mode                       // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

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

6.3.5 GrB_BinaryOp_free: free a user-defined binary operator

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

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

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

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

6.3.6 ANY and PAIR (ONEB) operators

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

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

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

The `ANY` function is associative and commutative, and can thus serve as an operator for a monoid. The selection of x are y is not randomized. Instead, SuiteSparse:GraphBLAS uses this freedom to compute as fast a result as possible. When used as the monoid in a dot product,

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

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

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

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

6.4 GraphBLAS IndexUnaryOp operators: GrB_IndexUnaryOp

An index-unary operator is a scalar function of the form $z = f(a_{ij}, i, j, y)$ that is applied to the entries a_{ij} of an m-by-n matrix. It can be used in GrB_apply (Section 12.12) or in GrB_select (Section 12.13) to select entries from a matrix or vector.

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

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

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

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

The following methods operate on the GrB_IndexUnaryOp object:

GraphBLAS function	purpose	Section
GrB_IndexUnaryOp_new	create a user-defined index-unary operator	6.4.1
GxB_IndexUnaryOp_new	create a named user-defined index-unary operator	6.4.2
GrB_IndexUnaryOp_wait	wait for a user-defined index-unary operator	6.4.3
GrB_get	get properties of an operator	10.5
GrB_set	set the operator name/definition	10.5
GrB_IndexUnaryOp_free	free a user-defined index-unary operator	6.4.4

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

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

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

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

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

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

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

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

Creates a named `GrB_IndexUnaryOp`. Only the first 127 characters of `idxop_name` are used. The `ixdop_defn` is a string containing the entire function itself.

The two strings `idxop_name` and `idxop_defn` are optional, but are required to enable the JIT compilation of kernels that use this operator. The strings can also be set the `GrB_set` after the operator is created with `GrB_IndexUnaryOp_new`. For example:

```
void banded_idx
(
    bool *z,
    const int64_t *x,   // unused
    int64_t i,
    int64_t j,
    const int64_t *thunk
)
{
    // d = abs (j-i)
    int64_t d = j-i ;
    if (d < 0) d = -d ;
    (*z) = (d <= *thunk) ;
}

#define BANDED_IDX_DEFN                      \
"void banded_idx                         \n" \
"(                                       \n" \
"    bool *z,                            \n" \
"    const int64_t *x,   // unused       \n" \
"    int64_t i,                          \n" \
"    int64_t j,                          \n" \
```

```
"    const int64_t *thunk              \n" \
")                                     \n" \
"{                                     \n" \
"    int64_t d = j-i ;                 \n" \
"    if (d < 0) d = -d ;               \n" \
"    (*z) = (d <= *thunk) ;            \n" \
"}"

GxB_IndexUnaryOp_new (&Banded,
    (GxB_index_unary_function) banded_idx,
    GrB_BOOL, GrB_INT64, GrB_INT64,
    "banded_idx", BANDED_IDX_DEFN)) ;
```

If JIT compilation is enabled, or if the corresponding JIT kernel has been copied into the `PreJIT` folder, the `function` may be `NULL`. In this case, a JIT kernel is compiled that contains just the user-defined function. If the JIT is disabled and the `function` is `NULL`, this method returns `GrB_NULL_POINTER`.

The above example is identical to the following usage except that `GrB_IndexUnaryOp_new` requires a non-NULL function pointer. The `banded_idx` function is defined the same as above.

```
void banded_idx ... see above
#define BANDED_IDX_DEFN  ... see above

GrB_IndexUnaryOp_new (&Banded,
    (GxB_index_unary_function) banded_idx,
    GrB_BOOL, GrB_INT64, GrB_INT64) ;
GrB_set (Banded, "banded_idx", GxB_JIT_C_NAME)) ;
GrB_set (Banded, BANDED_IDX_DEFN, GxB_JIT_C_DEFINITION)) ;
```

6.4.3 GrB_IndexUnaryOp_wait: wait for an index-unary operator

```
GrB_Info GrB_wait               // wait for a user-defined binary operator
(
    GrB_IndexUnaryOp op,        // index-unary operator to wait for
    int mode                    // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

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

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

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

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

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

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

6.5 GraphBLAS index-binary operators: GxB_IndexBinaryOp

An index-binary operator is a scalar function of the following form:

$$z = f(x, i_x, j_x, y, i_y, j_y, \Theta),$$

where the value x appears at row i_x and column j_x in its matrix, and the value y appears at row i_y and column j_y in its matrix. The value Θ is a scalar that is the same for all uses of the operator. See our IEEE HPEC'24 paper for more details ([MBM+24]).

When used in an element-wise method for $\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$ and related methods (`GrB_eWiseAdd`, `GxB_eWiseUnion`, `GrB_eWiseMult`, or `GrB_Kronecker`), the operator is used for a pair of entries a_{ij} and b_{ij}, as

$$z = f(a_{ij}, i, j, b_{ij}, i, j, \Theta).$$

When used as the multiplicative operator in a semiring, to compute $\mathbf{C} = \mathbf{A} \oplus . \otimes \mathbf{B}$, the operator is used as

$$z = f(a_{ik}, i, k, b_{kj}, k, j, \Theta)$$

to compute an entry to be summed by the monoid of the semiring.

No GraphBLAS operations directly use the `GxB_IndexBinaryOp`. Instead, the operator is coupled with a scalar `Theta` value to create a new index-based binary operator, which is simply a special case of a `GrB_BinaryOp`. The resulting `GrB_BinaryOp` can then be passed to element-wise methods and as the multiplicative operator of a new semiring.

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

```
void f
(
    void *z,            // output value z, of type ztype
    const void *x,      // input value x of type xtype; value of v(ix) or A(ix,jx)
    GrB_Index ix,       // row index of v(ix) or A(ix,jx)
    GrB_Index jx,       // column index of A(ix,jx), or zero for v(ix)
    const void *y,      // input value y of type ytype; value of w(iy) or B(iy,jy)
    GrB_Index iy,       // row index of w(iy) or B(iy,jy)
    GrB_Index jy,       // column index of B(iy,jy), or zero for w(iy)
    const void *theta   // input scalar theta of type theta_type
) ;
```

The following binary operators (GrB_BinaryOp objects) are pre-defined, where N can be INT32 or INT64. These operators do not use theta. Instead, the offset of 1 in GxB_FIRSTI1 is fixed into the operator itself.

Built-in index-based binary operators for any type			
GraphBLAS name	types (domains)	$z = f(x, y)$	description
GxB_FIRSTI_N	$\rightarrow N$	$z = i_x$	row index of x (0-based)
GxB_FIRSTI1_N	$\rightarrow N$	$z = i_x + 1$	row index of x (1-based)
GxB_FIRSTJ_N	$\rightarrow N$	$z = j_x$	column index of x (0-based)
GxB_FIRSTJ1_N	$\rightarrow N$	$z = j_x + 1$	column index of x (1-based)
GxB_SECONDI_N	$\rightarrow N$	$z = i_y$	row index of y (0-based)
GxB_SECONDI1_N	$\rightarrow N$	$z = i_y + 1$	row index of y (1-based)
GxB_SECONDJ_N	$\rightarrow N$	$z = j_y$	column index of y (0-based)
GxB_SECONDJ1_N	$\rightarrow N$	$z = j_y + 1$	column index of y (1-based)

The following methods operate on the GxB_IndexBinaryOp object:

GraphBLAS function	purpose	Section
GxB_IndexBinaryOp_new	create a named user-defined index-binary operator	6.5.1
GxB_IndexBinaryOp_wait	wait for a user-defined index-binary operator	6.5.2
GrB_get	get properties of an operator	10.7
GrB_set	set the operator name/definition	10.7
GxB_IndexBinaryOp_free	free a user-defined index-binary operator	6.5.3
GxB_BinaryOp_new_IndexOp	create a new index-based GrB_BinaryOp	6.5.4

6.5.1 GxB_IndexBinaryOp_new: create a user-defined index-binary operator

```
GrB_Info GxB_IndexBinaryOp_new
(
    GxB_IndexBinaryOp *op,          // handle for the new index binary operator
    GxB_index_binary_function function, // pointer to the index binary function
    GrB_Type ztype,                 // type of output z
    GrB_Type xtype,                 // type of input x
    GrB_Type ytype,                 // type of input y
    GrB_Type theta_type,            // type of input theta
    const char *idxbinop_name,      // name of the user function
    const char *idxbinop_defn       // definition of the user function
) ;
```

Creates a named `GxB_IndexBinaryOp`. Only the first 127 characters of `idxbinop_name` are used. The `ixdbinop_defn` is a string containing the entire function itself.

The two strings `idxbinop_name` and `idxbinop_defn` are optional, but are required to enable the JIT compilation of kernels that use this operator. For example, the following operator can be used to compute the argmax of a matrix with a single call to `GrB_mxv`. It returns a vector `c` where `c(i) = (k,v)`, where the largest value in the ith row of `A` has value `v` and appears in column `k`. If multiple values in the ith row have the same largest value, the one with the smallest column index is returned.

```
typedef struct { int64_t k ; double v ; } tuple_fp64 ;
#define FP64_K "typedef struct { int64_t k ; double v ; } tuple_fp64 ;"
void make_fp64 (tuple_fp64 *z,
    const double *x, GrB_Index ix, GrB_Index jx,
    const void   *y, GrB_Index iy, GrB_Index jy,
    const void *theta)
{
    z->k = (int64_t) jx ;
    z->v = (*x) ;
}
void max_fp64 (tuple_fp64 *z, const tuple_fp64 *x, const tuple_fp64 *y)
{
    if (x->v > y->v || (x->v == y->v && x->k < y->k))
    {
        z->k = x->k ;
        z->v = x->v ;
    }
    else
```

```
    {
        z->k = y->k ;
        z->v = y->v ;
    }
}
#define MAX_FP64 (a string containing the max_fp64 function above)

// create the types and operators:
GrB_Scalar Theta ;                  // unused, but cannot be NULL
GrB_Scalar_new (&Theta, GrB_BOOL) ;
GrB_Scalar_setElement_BOOL (Theta, 0) ;
GxB_IndexBinaryOp Iop ;
GrB_BinaryOp Bop, MonOp ;
GrB_Type Tuple ;
GxB_Type_new (&Tuple, sizeof (tuple_fp64), "tuple_fp64", FP64_K) ;
GxB_IndexBinaryOp_new (&Iop, make_fp64, Tuple, GrB_FP64, GrB_BOOL, GrB_BOOL,
    "make_fp64", MAKE_FP64)) ;
GxB_BinaryOp_new_IndexOp (&Bop, Iop, Theta) ;
tuple_fp64 id ;
memset (&id, 0, sizeof (tuple_fp64)) ;
id.k = INT64_MAX ;
id.v = (double) (-INFINITY) ;
GxB_BinaryOp_new (&MonOp, max_fp64, Tuple, Tuple, Tuple, "max_fp64", MAX_FP64) ;
GrB_Monoid MonOp ;
GrB_Semiring Semiring ;
GrB_Monoid_new_UDT (&Monoid, MonOp, &id) ;
GrB_Semiring_new (&Semiring, Monoid, Bop) ;

// compute the argmax of each row of a GrB_FP64 matrix A:
// y = zeros (ncols,1) ;
GrB_Vector y ;
GrB_Matrix_new (&y, GrB_BOOL, ncols, 1)) ;
GrB_Matrix_assign_BOOL (y, NULL, NULL, 0, GrB_ALL, ncols, GrB_ALL, 1, NULL)) ;
// c = A*y using the argmax semiring
GrB_Vector_new (&c, Tuple, nrows, 1)) ;
GrB_mxv (c, NULL, NULL, Semiring, A, y, NULL) ;
```

6.5.2 GxB_IndexBinaryOp_wait: wait for an index-binary operator

```
GrB_Info GxB_IndexBinaryOp_wait
(
    GxB_IndexBinaryOp op,
    int mode                    // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

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

6.5.3 GxB_IndexBinaryOp_free: free a user-defined index-binary operator

```
GrB_Info GrB_free               // free a user-created index-binary operator
(
    GxB_IndexBinaryOp *op       // handle of IndexBinaryOp to free
) ;
```

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

```
GxB_IndexBinaryOp_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. No built-in index-binary operators exist, but if they did, the method does nothing at all if passed a built-in index-binary operator.

6.5.4 GxB_BinaryOp_new_IndexOp: create a index-based binary operator

```
GrB_Info GxB_BinaryOp_new_IndexOp
(
    GrB_BinaryOp *binop,            // handle of binary op to create
    GxB_IndexBinaryOp idxbinop,     // based on this index binary op
    GrB_Scalar theta                // theta value to bind to the new binary op
) ;
```

The `GxB_IndexBinaryOp` cannot be directly used in any GraphBLAS operation such as `GrB_mxm`. Instead, it must be used to create a new index-based `GrB_BinaryOp`. The resulting binary operator can then be used to as the multiplicative operator in a new user-defined semiring, or as the primary binary operator of the element-wise operations (`eWiseAdd`, `eWiseUnion`, `eWiseMult`, or `kronecker`).

The resulting binary operator cannot be used as the `accum` operator in any GraphBLAS operation. It also cannot be used in other places where a binary operator appears, including `GrB_*_build`, `GrB_apply`, `GrB_reduce` and `GrB_*_sort`.

The `GxB_BinaryOp_new_IndexOp` method creates this index-based binary operator. It takes two input parameters: an index-binary operator, and a scalar `Theta`. The value of `Theta` is copied into this new binary operator, and the value cannot be changed. To change `Theta`, the binary operator must be freed, and any semiring that would like to use the new value of `Theta` must also be recreated.

An example of its use is given in Section 6.5.1.

6.6 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 operator must also have an identity element, or "zero" in this domain, such that $f(x, 0) = f(0, x) = x$. Recall that an associative operator $f(x, y)$ is one for which the condition $f(a, f(b, c)) = f(f(a, b), c)$ always holds. That is, operator can be applied in any order and the results remain the same. If used in a semiring, the operator must also be commutative.

The 77 predefined monoids are listed in the table below, which includes nearly all monoids that can be constructed from built-in binary operators. A few additional monoids can be defined with `GrB_Monoid_new` using built-in operators, such as bitwise monoids for signed integers. Recall that T denotes any built-in type (including boolean, integer, floating point real, and complex), R denotes any non-complex type (including bool), I denotes any integer type, and Z denotes any complex type. Let S denote the 10 non-boolean real types. Let U denote all unsigned integer types.

The table lists the GraphBLAS monoid, its type, expression, identity value, and *terminal* value (if any). For these built-in monoids, 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_MONOID_UINT8` has an identity of 255 and a terminal value of 0.

When computing with a monoid, the computation can terminate early if the terminal value arises. No further work is needed since the result will not change. This value is called the terminal value instead of the annihilator, since a user-defined operator can be created with a terminal value that is not an annihilator. See Section 6.6.3 for an example.

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

GraphBLAS operator	types (domains)	expression $z = f(x, y)$	identity	terminal
GrB_PLUS_MONOID_S	$S \times S \to S$	$z = x + y$	0	none
GrB_TIMES_MONOID_S	$S \times S \to S$	$z = xy$	1	0 or none (see note)
GrB_MIN_MONOID_S	$S \times S \to S$	$z = \min(x, y)$	$+\infty$	$-\infty$
GrB_MAX_MONOID_S	$S \times S \to S$	$z = \max(x, y)$	$-\infty$	$+\infty$
GxB_PLUS_Z_MONOID	$Z \times Z \to Z$	$z = x + y$	0	none
GxB_TIMES_Z_MONOID	$Z \times Z \to Z$	$z = xy$	1	none
GxB_ANY_T_MONOID	$T \times T \to T$	$z = x$ or y	any	any
GrB_LOR_MONOID	bool $\times$ bool $\to$ bool	$z = x \vee y$	false	true
GrB_LAND_MONOID	bool $\times$ bool $\to$ bool	$z = x \wedge y$	true	false
GrB_LXOR_MONOID	bool $\times$ bool $\to$ bool	$z = x \veebar y$	false	none
GrB_LXNOR_MONOID	bool $\times$ bool $\to$ bool	$z = (x == y)$	true	none
GxB_BOR_U_MONOID	$U \times U \to U$	`z=x|y`	all bits zero	all bits one
GxB_BAND_U_MONOID	$U \times U \to U$	`z=x&y`	all bits one	all bits zero
GxB_BXOR_U_MONOID	$U \times U \to U$	`z=x^y`	all bits zero	none
GxB_BXNOR_U_MONOID	$U \times U \to U$	`z=~(x^y)`	all bits one	none

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`).

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

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

GraphBLAS function	purpose	Section
`GrB_Monoid_new`	create a user-defined monoid	6.6.1
`GrB_Monoid_wait`	wait for a user-defined monoid	6.6.2
`GxB_Monoid_terminal_new`	create a monoid that has a terminal value	6.6.3
`GrB_get`	get properties of a monoid	10.8
`GrB_set`	set the monoid name	10.8
`GrB_Monoid_free`	free a monoid	6.6.4

6.6.1 GrB_Monoid_new: create a monoid

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

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

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

If `op` is a built-in operator with a known identity value, then the `identity` parameter is ignored, and its known identity value is used instead. The `op` cannot be a binary operator created by `GxB_BinaryOp_new_IndexOp`.

6.6.2 GrB_Monoid_wait: wait for a monoid

```
GrB_Info GrB_wait               // wait for a user-defined monoid
(
    GrB_Monoid monoid,          // monoid to wait for
    int mode                    // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

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

6.6.3 GxB_Monoid_terminal_new: create a monoid with terminal

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

GxB_Monoid_terminal_new is identical to GrB_Monoid_new, except that it allows for the specification of a *terminal value*. The <type> of the terminal value is the same as the identity parameter; see Section 6.6.1 for details.

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

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

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

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

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

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

6.6.4 GrB_Monoid_free: free a monoid

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

GrB_Monoid_frees frees a monoid. Either usage:

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

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

6.7 GraphBLAS semirings: GrB_Semiring

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

GraphBLAS function	purpose	Section
`GrB_Semiring_new`	create a user-defined semiring	6.7.1
`GrB_Semiring_wait`	wait for a user-defined semiring	6.7.2
`GrB_get`	get properties of a semiring	10.9
`GrB_set`	set the semiring name	10.9
`GrB_Semiring_free`	free a semiring	6.7.3

6.7.1 GrB_Semiring_new: create a semiring

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

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

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

MAX_MIN. Four semirings are for boolean types only: LOR_LAND, LAND_LOR, LXOR_LAND, and LXNOR_LOR.

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

- 1000 semirings with a multiplier $T \times T \rightarrow T$ where T is any of the 10 non-Boolean, real types, from the complete cross product of:
  - 5 monoids (MIN, MAX, PLUS, TIMES, ANY)
  - 20 multiply operators (FIRST, SECOND, PAIR (same as ONEB), MIN, MAX, PLUS, MINUS, RMINUS, TIMES, DIV, RDIV, ISEQ, ISNE, ISGT, ISLT, ISGE, ISLE, LOR, LAND, LXOR).
  - 10 non-Boolean types, T
- 300 semirings with a comparator $T \times T \rightarrow$ bool, where T is non-Boolean and real, from the complete cross product of:
  - 5 Boolean monoids (LAND, LOR, LXOR, EQ, ANY)
  - 6 multiply operators (EQ, NE, GT, LT, GE, LE)
  - 10 non-Boolean types, T
- 55 semirings with purely Boolean types, bool $\times$ bool $\rightarrow$ bool, from the complete cross product of:
  - 5 Boolean monoids (LAND, LOR, LXOR, EQ, ANY)
  - 11 multiply operators (FIRST, SECOND, PAIR (same as ONEB), LOR, LAND, LXOR, EQ, GT, LT, GE, LE)
- 54 complex semirings, $Z \times Z \rightarrow Z$ where Z is GxB_FC32 (single precision complex) or GxB_FC64 (double precision complex):
  - 3 complex monoids (PLUS, TIMES, ANY)
  - 9 complex multiply operators (FIRST, SECOND, PAIR (same as ONEB), PLUS, MINUS, TIMES, DIV, RDIV, RMINUS)
  - 2 complex types, Z

- 64 bitwise semirings, $U \times U \rightarrow U$ where U is an unsigned integer.
  - 4 bitwise monoids (BOR, BAND, BXOR, BXNOR)
  - 4 bitwise multiply operators (the same list)
  - 4 unsigned integer types
- 80 index-based semirings, $X \times X \rightarrow N$ where N is INT32 or INT64:
  - 5 monoids (MIN, MAX, PLUS, TIMES, ANY)
  - 8 index-based operators (FIRSTI, FIRSTI1, FIRSTJ, FIRSTJ1, SECONDI, SECONDI1, SECONDJ, SECONDJ1)
  - 2 integer types (INT32, INT64)

The multiply operator can be any a binary operator, including one created by GxB_BinaryOp_new_IndexOp.

6.7.2 GrB_Semiring_wait: wait for a semiring

```
GrB_Info GrB_wait               // wait for a user-defined semiring
(
    GrB_Semiring semiring,      // semiring to wait for
    int mode                    // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

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

6.7.3 GrB_Semiring_free: free a semiring

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

GrB_Semiring_free frees a semiring. Either usage:

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

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

6.8 GraphBLAS scalars: GrB_Scalar

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

GraphBLAS function	purpose	Section
`GrB_Scalar_new`	create a scalar	6.8.1
`GrB_Scalar_wait`	wait for a scalar	6.8.2
`GrB_Scalar_dup`	copy a scalar	6.8.3
`GrB_Scalar_clear`	clear a scalar of its entry	6.8.4
`GrB_Scalar_nvals`	return number of entries in a scalar	6.8.5
`GrB_get`	get properties of a scalar	10.12
`GrB_set`	set properties of a scalar	10.12
`GrB_Scalar_setElement`	set the single entry of a scalar	6.8.6
`GrB_Scalar_extractElement`	get the single entry from a scalar	6.8.7
`GxB_Scalar_memoryUsage`	memory used by a scalar	6.8.8
`GxB_Scalar_type`	type of a scalar	6.8.9
`GrB_Scalar_free`	free a scalar	6.8.10

6.8.1 GrB_Scalar_new: create a scalar

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

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

6.8.2 GrB_Scalar_wait: wait for a scalar

```
GrB_Info GrB_wait               // wait for a scalar
(
    GrB_Scalar s,               // scalar to wait for
    int mode                    // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

In non-blocking mode, the computations for a `GrB_Scalar` may be delayed. In this case, the scalar is not yet safe to use by multiple independent user threads. A user application may force completion of a scalar s via `GrB_Scalar_wait(&s)` (in v5.2.0), or `GrB_Scalar_wait(s,mode)` (in v6.0.0). With a `mode` of `GrB_MATERIALIZE`, all pending computations are finished, and different user threads may simultaneously call GraphBLAS operations that use the scalar `s` as an input parameter. See Section 10.2.2 if GraphBLAS is compiled without OpenMP.

6.8.3 GrB_Scalar_dup: copy a scalar

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

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

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

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

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

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

6.8.4 GrB_Scalar_clear: clear a scalar of its entry

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

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

6.8.5 GrB_Scalar_nvals: return the number of entries in a scalar

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

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

6.8.6 GrB_Scalar_setElement: set the single entry of a scalar

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

`GrB_Scalar_setElement` sets the single entry in a scalar, like `s = sparse(x)` in MATLAB notation. For further details of this function, see `GrB_Matrix_setElement` in Section 6.10.10. If an error occurs, `GrB_error(&err,s)` returns details about the error. The scalar `x` can be any non-opaque C scalar corresponding to a built-in type, or `void *` for a user-defined type. It cannot be a `GrB_Scalar`.

6.8.7 GrB_Scalar_extractElement: get the single entry from a scalar

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

GrB_Scalar_extractElement extracts the single entry from a sparse scalar, like x = full(s) in MATLAB. Further details of this method are discussed in Section 6.10.11, which discusses GrB_Matrix_extractElement. **NOTE:** if no entry is present in the scalar s, then x is not modified, and the return value of GrB_Scalar_extractElement is GrB_NO_VALUE.

6.8.8 GxB_Scalar_memoryUsage: memory used by a scalar

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

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

6.8.9 GxB_Scalar_type: type of a scalar

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

Returns the type of a scalar. See GxB_Matrix_type for details (Section 6.10.23).

6.8.10 GrB_Scalar_free: free a scalar

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

GrB_Scalar_free frees a scalar. Either usage:

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

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

6.9 GraphBLAS vectors: GrB_Vector

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

GraphBLAS function	purpose	Section
`GrB_Vector_new`	create a vector	6.9.1
`GrB_Vector_wait`	wait for a vector	6.9.2
`GrB_Vector_dup`	copy a vector	6.9.3
`GrB_Vector_clear`	clear a vector of all entries	6.9.4
`GrB_Vector_size`	size of a vector	6.9.5
`GrB_Vector_nvals`	number of entries in a vector	6.9.6
`GrB_get`	get properties of a vector	10.11
`GrB_set`	set properties of a vector	10.11
`GrB_Vector_build`	build a vector from tuples	6.9.7
`GxB_Vector_build_Scalar`	build a vector from tuples	6.9.8
`GrB_Vector_setElement`	add an entry to a vector	6.9.9
`GrB_Vector_extractElement`	get an entry from a vector	6.9.10
`GxB_Vector_isStoredElement`	check if entry present in vector	6.9.11
`GrB_Vector_removeElement`	remove an entry from a vector	6.9.12
`GrB_Vector_extractTuples`	get all entries from a vector	6.9.13
`GrB_Vector_resize`	resize a vector	6.9.14
`GxB_Vector_diag`	extract a diagonal from a matrix	6.9.15
`GxB_Vector_memoryUsage`	memory used by a vector	6.9.16
`GxB_Vector_type`	type of the matrix	6.9.17
`GrB_Vector_free`	free a vector	6.9.18
`GxB_Vector_serialize`	serialize a vector	6.11.1
`GxB_Vector_deserialize`	deserialize a vector	6.11.2
`GxB_Vector_sort`	sort a vector	6.13.1

Refer to Section 6.11 for serialization/deserialization methods and to Section 6.13 for sorting methods.

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

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

6.9.2 GrB_Vector_wait: wait for a vector

```
GrB_Info GrB_wait               // wait for a vector
(
    GrB_Vector w,               // vector to wait for
    int mode                    // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

In non-blocking mode, the computations for a `GrB_Vector` may be delayed. In this case, the vector is not yet safe to use by multiple independent user threads. A user application may force completion of a vector `w` via `GrB_Vector_wait(&w)` (in v5.2.0), or `GrB_Vector_wait(w,mode)` (in v6.0.0). With a `mode` of `GrB_MATERIALIZE`, all pending computations are finished, and different user threads may simultaneously call GraphBLAS operations that use the vector `w` as an input parameter. See Section 10.2.2 if GraphBLAS is compiled without OpenMP.

6.9.3 GrB_Vector_dup: copy a vector

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

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

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

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

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

Then w and u are two different vectors that currently have the same set of values, but they do not depend on each other. Modifying one has no effect on the other. The GrB_NAME is copied into the new vector.

6.9.4 GrB_Vector_clear: clear a vector of all entries

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

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

6.9.5 GrB_Vector_size: return the size of a vector

```
GrB_Info GrB_Vector_size    // get the dimension of a vector
(
    GrB_Index *n,           // vector dimension is n-by-1
    const GrB_Vector v      // vector to query
) ;
```

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

6.9.6 GrB_Vector_nvals: return the number of entries in a vector

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

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

6.9.7 GrB_Vector_build: build a vector from a set of tuples

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

`GrB_Vector_build` constructs a sparse vector `w` from a set of tuples, `I` and `X`, each of length `nvals`. The vector `w` must have already been initialized with `GrB_Vector_new`, and it must have no entries in it before calling `GrB_Vector_build`. This function is just like `GrB_Matrix_build` (see Section 6.10.8), except that it builds a sparse vector instead of a sparse matrix. For a description of what `GrB_Vector_build` does, refer to `GrB_Matrix_build`. For a vector, the list of column indices `J` in `GrB_Matrix_build` is implicitly a vector of length `nvals` all equal to zero. Otherwise the methods are identical.

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

SPEC: Results are defined even if `dup` is non-associative and/or non-commutative.

6.9.8 GxB_Vector_build_Scalar: build a vector from a set of tuples

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

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

6.9.9 GrB_Vector_setElement: add an entry to a vector

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

`GrB_Vector_setElement` sets a single entry in a vector, `w(i) = x`. The operation is exactly like setting a single entry in an `n`-by-1 matrix, `A(i,0) = x`, where the column index for a vector is implicitly `j=0`. For further details of this function, see `GrB_Matrix_setElement` in Section 6.10.10. If an error occurs, `GrB_error(&err,w)` returns details about the error.

6.9.10 GrB_Vector_extractElement: get an entry from a vector

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

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

`GrB_Vector_extractElement` extracts a single entry from a vector, `x = v(i)`. The method is identical to extracting a single entry `x = A(i,0)` from an `n`-by-1 matrix; see Section 6.10.11.

6.9.11 GxB_Vector_isStoredElement: check if entry present in vector

```
GrB_Info GxB_Vector_isStoredElement
(
    const GrB_Vector v,         // check presence of entry v(i)
    GrB_Index i                 // index
) ;
```

GxB_Vector_isStoredElement checks if a single entry v(i) is present, returning GrB_SUCCESS if the entry is present or GrB_NO_VALUE otherwise. The value of v(i) is not returned. See also Section 6.10.12.

6.9.12 GrB_Vector_removeElement: remove an entry from a vector

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

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

6.9.13 GrB_Vector_extractTuples: get all entries from a vector

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

GrB_Vector_extractTuples extracts all tuples from a sparse vector, analogous to [I,~,X] = find(v) in MATLAB/Octave. This function is identical to its GrB_Matrix_extractTuples counterpart, except that the array of column indices J does not appear in this function. Refer to Section 6.10.14 where further details of this function are described.

6.9.14 GrB_Vector_resize: resize a vector

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

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

6.9.15 GxB_Vector_diag: extract a diagonal from a matrix

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

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

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

6.9.16 GxB_Vector_memoryUsage: memory used by a vector

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

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

6.9.17 GxB_Vector_type: 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
) ;
```

Returns the type of a vector. See `GxB_Matrix_type` for details (Section 6.10.23).

6.9.18 GrB_Vector_free: free a vector

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

`GrB_Vector_free` frees a vector. Either usage:

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

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

6.10 GraphBLAS matrices: GrB_Matrix

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

GraphBLAS function	purpose	Section
`GrB_Matrix_new`	create a matrix	6.10.1
`GrB_Matrix_wait`	wait for a matrix	6.10.2
`GrB_Matrix_dup`	copy a matrix	6.10.3
`GrB_Matrix_clear`	clear a matrix of all entries	6.10.4
`GrB_Matrix_nrows`	number of rows of a matrix	6.10.5
`GrB_Matrix_ncols`	number of columns of a matrix	6.10.6
`GrB_Matrix_nvals`	number of entries in a matrix	6.10.7
`GrB_get`	get properties of a matrix	10.10
`GrB_set`	set properties of a matrix	10.10
`GrB_Matrix_build`	build a matrix from tuples	6.10.8
`GxB_Matrix_build_Scalar`	build a matrix from tuples	6.10.9
`GrB_Matrix_setElement`	add an entry to a matrix	6.10.10
`GrB_Matrix_extractElement`	get an entry from a matrix	6.10.11
`GxB_Matrix_isStoredElement`	check if entry present in matrix	6.10.12
`GrB_Matrix_removeElement`	remove an entry from a matrix	6.10.13
`GrB_Matrix_extractTuples`	get all entries from a matrix	6.10.14
`GrB_Matrix_resize`	resize a matrix	6.10.15
`GxB_Matrix_concat`	concatenate matrices	6.10.18
`GxB_Matrix_split`	split a matrix into matrices	6.10.19
`GrB_Matrix_diag`	diagonal matrix from vector	6.10.20
`GxB_Matrix_diag`	diagonal matrix from vector	6.10.21
`GxB_Matrix_memoryUsage`	memory used by a matrix	6.10.22
`GxB_Matrix_type`	type of the matrix	6.10.23
`GrB_Matrix_free`	free a matrix	6.10.24
`GrB_Matrix_serializeSize`	return size of serialized matrix	6.11.3
`GrB_Matrix_serialize`	serialize a matrix	6.11.4
`GxB_Matrix_serialize`	serialize a matrix	6.11.5
`GrB_Matrix_deserialize`	deserialize a matrix	6.11.6
`GxB_Matrix_deserialize`	deserialize a matrix	6.11.7

GraphBLAS function	purpose	Section
`GrB_Matrix_import`	import in various formats	6.12.1
`GrB_Matrix_export`	export in various formats	6.12.2
`GrB_Matrix_exportSize`	array sizes for export	6.12.3
`GrB_Matrix_exportHint`	hint best export format	6.12.4
`GxB_Matrix_sort`	sort a matrix	6.13.2

Refer to Section 6.11 for serialization/deserialization methods, Section 6.12

for GrB import/export methods, and Section 6.13 for sorting methods.

6.10.1 GrB_Matrix_new: create a matrix

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

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

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

Once a matrix is created, its format (by-row or by-column) can be arbitrarily changed with `GrB_set (A, fmt, GrB_STORAGE_ORIENTATION_HINT)` with `fmt` equal to `GrB_COLMAJOR` or `GrB_ROWMAJOR`.

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

6.10.2 GrB_Matrix_wait: wait for a matrix

```
GrB_Info GrB_wait               // wait for a matrix
(
    GrB_Matrix C,               // matrix to wait for
    int mode                    // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

In non-blocking mode, the computations for a `GrB_Matrix` may be delayed. In this case, the matrix is not yet safe to use by multiple independent user threads. A user application may force completion of a matrix `C` via `GrB_Matrix_wait(&C)` (in v5.2.0), or `GrB_Matrix_wait(C,mode)` (in v6.0.0). With a `mode` of `GrB_MATERIALIZE`, all pending computations are finished, and different user threads may simultaneously call GraphBLAS operations that use the matrix `C` as an input parameter. See Section 10.2.2 if GraphBLAS is compiled without OpenMP.

6.10.3 GrB_Matrix_dup: copy a matrix

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

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

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

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

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

Then C and A are two different matrices that currently have the same set of values, but they do not depend on each other. Modifying one has no effect on the other. The GrB_NAME is copied into the new matrix.

6.10.4 GrB_Matrix_clear: clear a matrix of all entries

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

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

6.10.5 GrB_Matrix_nrows: return the number of rows of a matrix

```
GrB_Info GrB_Matrix_nrows   // get the number of rows of a matrix
(
    GrB_Index *nrows,       // matrix has nrows rows
    const GrB_Matrix A      // matrix to query
) ;
```

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

6.10.6 GrB_Matrix_ncols: return the number of columns of a matrix

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

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

6.10.7 GrB_Matrix_nvals: return the number of entries in a matrix

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

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

6.10.8 GrB_Matrix_build: build a matrix from a set of tuples

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

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

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

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

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

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

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 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. The dup operator cannot be a binary operator created by GxB_BinaryOp_new_IndexOp.

6.10.9 GxB_Matrix_build_Scalar: build a matrix from a set of tuples

```
GrB_Info GxB_Matrix_build_Scalar    // build a matrix from (I,J,scalar) tuples
(
    GrB_Matrix C,                   // matrix to build
    const GrB_Index *I,             // array of row indices of tuples
    const GrB_Index *J,             // array of column indices of tuples
    GrB_Scalar scalar,              // value for all tuples
    GrB_Index nvals                 // number of tuples
) ;
```

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

6.10.10 GrB_Matrix_setElement: add an entry to a matrix

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

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

An error is returned (GrB_INVALID_INDEX) if the row index i is greater than or equal to the number of rows of C, or if the column index j is greater than or equal to the number of columns of C. Note that this error code differs from the same kind of condition in GrB_Matrix_build, which returns GrB_INDEX_OUT_OF_BOUNDS. This is because GrB_INVALID_INDEX is an

API error, and is caught immediately even in non-blocking mode, whereas GrB_INDEX_OUT_OF_BOUNDS is an execution error whose detection may wait until the computation completes sometime later.

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

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

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

A few methods and operations can be intermixed with setElement, in particular, some forms of the GrB_assign and GxB_subassign operations are compatible with the pending updates from setElement. Section 12.11 gives more details on which GxB_subassign and GrB_assign operations can be interleaved with calls to setElement without forcing updates to be assembled. Other methods that do not access the existing entries may also be done without forcing the updates to be assembled, namely GrB_Matrix_clear (which erases all pending updates), GrB_Matrix_free, GrB_Matrix_ncols, GrB_Matrix_nrows, GrB_get, 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 17.2 for an example of how to use GrB_Matrix_setElement. If an error occurs, GrB_error(&err,C) returns details about the error.

6.10.11 GrB_Matrix_extractElement: get an entry from a matrix

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

`GrB_Matrix_extractElement` extracts a single entry from a matrix `x=A(i,j)`. An error is returned (`GrB_INVALID_INDEX`) if the row index `i` is greater than or equal to the number of rows of `C`, or if column index `j` is greater than or equal to the number of columns of `C`. If the entry is present, `x=A(i,j)` is performed and the scalar `x` is returned with this value. The method returns `GrB_SUCCESS`. If no entry is present at `A(i,j)`, and `x` is a non-opaque C scalar, then `x` is not modified, and the return value of `GrB_Matrix_extractElement` is `GrB_NO_VALUE`. If `x` is a `GrB_Scalar`, then `x` is returned as an empty scalar with no entry, and `GrB_SUCCESS` is returned.

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

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

6.10.12 GxB_Matrix_isStoredElement: check if entry present in matrix

```
GrB_Info GxB_Matrix_isStoredElement
(
    const GrB_Matrix A,         // check for A(i,j)
    GrB_Index i,                // row index
    GrB_Index j                 // column index
) ;
```

GxB_Matrix_isStoredElement check if the single entry A(i,j) is present in the matrix A. It returns GrB_SUCCESS if the entry is present, or GrB_NO_VALUE otherwise. The value of A(i,j) is not returned. It is otherwise identical to GrB_Matrix_extractElement.

6.10.13 GrB_Matrix_removeElement: remove an entry from a matrix

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

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

6.10.14 GrB_Matrix_extractTuples: get all entries from a matrix

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

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

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

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

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

6.10.15 GrB_Matrix_resize: resize a matrix

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

`GrB_Matrix_resize` changes the size of a matrix. If the dimensions decrease, entries that fall outside the resized matrix are deleted. Unlike `GxB_Matrix_reshape*` (see Sections 6.10.16 and 6.10.17), entries remain in their same position after resizing the matrix.

6.10.16 GxB_Matrix_reshape: **reshape a matrix**

```
GrB_Info GxB_Matrix_reshape     // reshape a GrB_Matrix in place
(
    // input/output:
    GrB_Matrix C,               // input/output matrix, reshaped in place
    // input:
    bool by_col,                // true if reshape by column, false if by row
    GrB_Index nrows_new,        // new number of rows of C
    GrB_Index ncols_new,        // new number of columns of C
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_reshape` changes the size of a matrix `C`, taking entries from the input matrix either column-wise or row-wise. If matrix `C` on input is `nrows`-by-`ncols`, and the requested dimensions of `C` on output are `nrows_new`-by-`nrows_cols`, then the condition `nrows*ncols == nrows_new*nrows_cols` must hold. The matrix `C` is modified in-place, as both an input and output for this method. To create a new matrix, use `GxB_Matrix_reshapeDup` instead (Section 6.10.17).

For example, if `C` is 3-by-4 on input, and is reshaped column-wise to have dimensions 2-by-6:

```
C on input      C on output (by_col true)
00 01 02 03     00 20 11 02 22 13
10 11 12 13     10 01 21 12 03 23
20 21 22 23
```

If the same `C` on input is reshaped row-wise to dimensions 2-by-6:

```
C on input      C on output (by_col false)
00 01 02 03     00 01 02 03 10 11
10 11 12 13     12 13 20 21 22 23
20 21 22 23
```

NOTE: because an intermediate linear index must be computed for each entry, `GxB_Matrix_reshape` cannot be used on matrices for which `nrows*ncols` exceeds 2^{60}.

6.10.17 GxB_Matrix_reshapeDup: reshape a matrix

```
GrB_Info GxB_Matrix_reshapeDup // reshape a GrB_Matrix into another GrB_Matrix
(
    // output:
    GrB_Matrix *C,              // newly created output matrix, not in place
    // input:
    GrB_Matrix A,               // input matrix, not modified
    bool by_col,                // true if reshape by column, false if by row
    GrB_Index nrows_new,        // number of rows of C
    GrB_Index ncols_new,        // number of columns of C
    const GrB_Descriptor desc
) ;
```

`GxB_Matrix_reshapeDup` is identical to `GxB_Matrix_reshape` (see Section 6.10.16), except that creates a new output matrix `C` that is reshaped from the input matrix `A`.

6.10.18 GxB_Matrix_concat: concatenate matrices

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

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

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

$$C = \left[\begin{array}{ccccc} A_{0,0} & A_{0,1} & A_{0,2} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & A_{1,2} & \cdots & A_{1,n-1} \\ \cdots & & & & \\ A_{m-1,0} & A_{m-1,1} & A_{m-1,2} & \cdots & A_{m-1,n-1} \end{array} \right]$$

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

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

6.10.19 GxB_Matrix_split: split a matrix

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

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

6.10.20 GrB_Matrix_diag: construct a diagonal matrix

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

`GrB_Matrix_diag` constructs a matrix from a vector. Let n be the length of the `v` vector, from `GrB_Vector_size (&n, v)`. If `k` $= 0$, then `C` is an

n-by-n diagonal matrix with the entries from v along the main diagonal of C, with C(i,i)=v(i). If k is nonzero, C is square with dimension $n + |k|$. If k is positive, it denotes diagonals above the main diagonal, with C(i,i+k)=v(i). If k is negative, it denotes diagonals below the main diagonal of C, with C(i-k,i)=v(i). This behavior is identical to the MATLAB statement C=diag(v,k), where v is a vector.

The output matrix C is a newly-constructed square matrix with the same type as the input vector v. No typecasting is performed.

6.10.21 GxB_Matrix_diag: build a diagonal matrix

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

Identical to GrB_Matrix_diag, except for the extra parameter (a descriptor to provide control over the number of threads used), and this method is not a constructor.

The matrix C must already exist on input, of the correct size. It must be square of dimension $n + |k|$ where the vector v has length n. Any existing entries in C are discarded. The type of C is preserved, so that if the type of C and v differ, the entries are typecasted into the type of C. Any settings made to C by GrB_set (format by row or by column, bitmap switch, hyper switch, and sparsity control) are unchanged.

6.10.22 GxB_Matrix_memoryUsage: memory used by a matrix

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

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

6.10.23 GxB_Matrix_type: type of a matrix

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

Returns the type of a matrix. The `type` parameter is not allocated. Calling `GxB_Matrix_type` is identical to making a shallow pointer copy of the type used to create a matrix. In particular, suppose a matrix is created, and a copy of its type is saved at the same time:

```
GrB_Matrix_new (&A, atype, m, n) ;
GrB_Type save_type = atype ;
```

Sometime later, while the matrix `A` and its type `atype` have not been freed, the following two code fragments are identical:

```
// using GxB_Matrix_type:
GrB_Type atype2 ;
GxB_Matrix_type (&atype2, A) ;
assert (atype2 == save_type) ;

// without GxB_Matrix_type:
GrB_Type atype2 = save_type ;
```

As a result, freeing `atype2` would be the same as freeing the original `atype`.

6.10.24 GrB_Matrix_free: free a matrix

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

GrB_Matrix_free frees a matrix. Either usage:

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

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

6.11 Serialize/deserialize methods

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

All versions of SuiteSparse:GraphBLAS that implement serialization/deserialization use essentially the same format for the blob, so the library versions are compatible with each other. Version v9.0.0 adds the `GrB_NAME` and `GrB_EL_TYPE_STRING` to the blob in an upward compatible manner, so that older versions of SS:GraphBLAS can read the blobs created by v9.0.0; they simply ignore those components.

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

By default, ZSTD (level 1) compression is used for serialization, but other options can be selected via the descriptor: `GrB_set (desc, method, GxB_COMPRESSION)`, where `method` is an integer selected from the following options:

method	description
`GxB_COMPRESSION_NONE`	no compression
`GxB_COMPRESSION_DEFAULT`	ZSTD, with default level 1
`GxB_COMPRESSION_LZ4`	LZ4
`GxB_COMPRESSION_LZ4HC`	LZ4HC, with default level 9
`GxB_COMPRESSION_ZSTD`	ZSTD, with default level 1

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

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

The ZSTD method can be specified as level 1 to 19, with 1 being the default. To compress with ZSTD at level 6, use:

```
GrB_set (desc, GxB_COMPRESSION_ZSTD + 6, GxB_COMPRESSION) ;
```

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

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

The following methods are described in this Section:

GraphBLAS function	purpose	Section
`GxB_Vector_serialize`	serialize a vector	6.11.1
`GxB_Vector_deserialize`	deserialize a vector	6.11.2
`GrB_Matrix_serializeSize`	return size of serialized matrix	6.11.3
`GrB_Matrix_serialize`	serialize a matrix	6.11.4
`GxB_Matrix_serialize`	serialize a matrix	6.11.5
`GrB_Matrix_deserialize`	deserialize a matrix	6.11.6
`GxB_Matrix_deserialize`	deserialize a matrix	6.11.7
`GrB_get`	get blob properties	10.15

6.11.1 GxB_Vector_serialize: serialize a vector

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

`GxB_Vector_serialize` serializes a vector into a single array of bytes (the blob), which is `malloc`'ed and filled with the serialized vector. By default, ZSTD (level 1) compression is used, but other options can be selected via the descriptor. Serializing a vector is identical to serializing a matrix; see Section 6.11.5 for more information.

6.11.2 GxB_Vector_deserialize: deserialize a vector

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

This method creates a vector `w` by deserializing the contents of the blob, constructed by `GxB_Vector_serialize`. Deserializing a vector is identical to deserializing a matrix; see Section 6.11.7 for more information.

The blob is allocated with the `malloc` function passed to `GxB_init`, or the C11 `malloc` if `GrB_init` was used to initialize GraphBLAS. The blob must be freed by the matching `free` method, either the `free` function passed to `GxB_init` or the C11 `free` if `GrB_init` was used.

6.11.3 GrB_Matrix_serializeSize: return size of serialized matrix

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

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

6.11.4 GrB_Matrix_serialize: serialize a matrix

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

`GrB_Matrix_serialize` serializes a matrix into a single array of bytes (the blob), which must be already allocated by the user application. On input, `&blob_size` is the size of the allocated blob in bytes. On output, it is reduced to the numbed of bytes actually used to serialize the matrix. After calling `GrB_Matrix_serialize`, the blob may be `realloc`'d to this revised size if desired (this is optional). ZSTD (level 1) compression is used to construct a compact blob.

6.11.5 GxB_Matrix_serialize: serialize a matrix

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

`GxB_Matrix_serialize` is identical to `GrB_Matrix_serialize`, except that it does not require a pre-allocated blob. Instead, it allocates the blob internally, and fills it with the serialized matrix. By default, ZSTD (level 1) compression is used, but other options can be selected via the descriptor.

The blob is allocated with the `malloc` function passed to `GxB_init`, or the C11 `malloc` if `GrB_init` was used to initialize GraphBLAS. The blob must be freed by the matching `free` method, either the `free` function passed to `GxB_init` or the C11 `free` if `GrB_init` was used.

6.11.6 GrB_Matrix_deserialize: deserialize a matrix

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

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

The `type` may be `NULL` if the blob holds a serialized matrix with a built-in type. In this case, the type is determined automatically. For user-defined types, the `type` must match the type of the matrix in the blob. The `GrB_get` method can be used to query the blob for the name of this type.

6.11.7 GxB_Matrix_deserialize: deserialize a matrix

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

Identical to `GrB_Matrix_deserialize`.

6.12 GraphBLAS import/export: using copy semantics

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

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

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

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

6.12.1 GrB_Matrix_import: import a matrix

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

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

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

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

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

- `GrB_CSR_FORMAT`: Compressed-row format. `Ap` is an array of size `nrows+1`. The arrays `Ai` and `Ax` are of size `nvals = Ap [nrows]`, and `Ap[0]` must be zero. The column indices of entries in the `i`th row appear in `Ai[Ap[i]...Ap[i+1]-1]`, and the values of those entries appear in the same locations in `Ax`. The column indices need not be in any particular order.
- `GrB_CSC_FORMAT`: Compressed-column format. `Ap` is an array of size `ncols+1`. The arrays `Ai` and `Ax` are of size `nvals = Ap [ncols]`, and `Ap[0]` must be zero. The row indices of entries in the `j`th column appear in `Ai[Ap[j]...Ap[j+1]-1]`, and the values of those entries appear in the same locations in `Ax`. The row indices need not be in any particular order.

- `GrB_COO_FORMAT`: Coordinate format. This is the same format as `GrB_Matrix_build`. The three arrays `Ap`, `Ai`, and `Ax` have the same size. The `k`th tuple has row index `Ai[k]`, column index `Ap[k]`, and value `Ax[k]`. The tuples can appear any order, but no duplicates are permitted.

6.12.2 GrB_Matrix_export: export a matrix

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

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

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

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

6.12.3 GrB_Matrix_exportSize: determine size of export

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

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

6.12.4 GrB_Matrix_exportHint: determine best export format

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

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

- `GxB_SPARSE`, `GrB_ROWMAJOR`: export as `GrB_CSR_FORMAT`
- `GxB_SPARSE`, `GrB_COLMAJOR`: export as `GrB_CSC_FORMAT`
- `GxB_HYPERSPARSE`: export as `GrB_COO_FORMAT`
- `GxB_BITMAP`, `GrB_ROWMAJOR`: export as `GrB_CSR_FORMAT`
- `GxB_BITMAP`, `GrB_COLMAJOR`: export as `GrB_CSC_FORMAT`
- `GxB_FULL`, `GrB_ROWMAJOR`: export as `GrB_CSR_FORMAT`
- `GxB_FULL`, `GrB_COLMAJOR`: export as `GrB_CSC_FORMAT`

6.13 Sorting methods

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

6.13.1 GxB_Vector_sort: sort a vector

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

GxB_Vector_sort is identical to sorting the single column of an n-by-1 matrix. Refer to Section 6.13.2 for details. The op cannot be a binary operator created by GxB_BinaryOp_new_IndexOp.

6.13.2 GxB_Matrix_sort: sort the rows/columns of a matrix

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

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

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

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

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

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

The outputs `C` and `P` are both optional; either one (but not both) may be `NULL`, in which case that particular output matrix is not computed. The `op` cannot be a binary operator created by `GxB_BinaryOp_new_IndexOp`.

6.14 GraphBLAS descriptors: GrB_Descriptor

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

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

```
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_AxB_METHOD = 1000, // descriptor for selecting C=A*B algorithm
    GxB_SORT = 35   // control sort in GrB_mxm
    GxB_COMPRESSION = 36,   // select compression for serialize
}
GrB_Desc_Field ;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  - GrB_DEFAULT means that a method is selected automatically.
  - GxB_AxB_SAXPY: select any saxpy-based method: GxB_AxB_GUSTAVSON, and/or GxB_AxB_HASH, or any mix of the two, in contrast to the dot-product method.
  - GxB_AxB_GUSTAVSON: an extended version of Gustavson's method [Gus78], which is a very good general-purpose method, but sometimes the workspace can be too large. Assuming all matrices are stored by column, it computes C(:,j)=A*B(:,j) with a sequence of *saxpy* operations (C(:,j)+=A(:,k)*B(k:,j) for each nonzero

B(k,j)). In the *coarse Gustavson* method, each internal thread requires workspace of size m, to the number of rows of C, which is not suitable if the matrices are extremely sparse or if there are many threads. For the *fine Gustavson* method, threads can share workspace and update it via atomic operations. If all matrices are stored by row, then it computes C(i,:)=A(i,:)*B in a sequence of sparse *saxpy* operations, and using workspace of size n per thread, or group of threads, corresponding to the number of columns of C.

  - GxB_AxB_HASH: a hash-based method, based on [NMAB18]. It is very efficient for hypersparse matrices, matrix-vector-multiply, and when $|\mathbf{B}|$ is small. SuiteSparse:GraphBLAS includes a *coarse hash* method, in which each thread has its own hash workspace, and a *fine hash* method, in which groups of threads share a single hash workspace, as concurrent data structure, using atomics.
  - GxB_AxB_DOT: computes C(i,j)=A(i,:)*B(j,:)', for each entry C(i,j). If the mask is present and not complemented, only entries for which M(i,j)=1 are computed. This is a very specialized method that works well only if the mask is present, very sparse, and not complemented, when C is small, or when C is bitmap or full. For example, it works very well when A and B are tall and thin, and C<M>=A*B' or C=A*B' are computed. These expressions assume all matrices are in CSR format. If in CSC format, then the dot-product method used for A'*B. The method is impossibly slow if C is large and the mask is not present, since it takes $\Omega(mn)$ time if C is m-by-n in that case. It does not use any workspace at all. Since it uses no workspace, it can work very well for extremely sparse or hypersparse matrices, when the mask is present and not complemented.

- GxB_SORT provides a hint to GrB_mxm, GrB_mxv, GrB_vxm, and GrB_reduce (to vector). These methods can leave the output matrix or vector in a jumbled state, where the final sort is left as pending work. This is typically fastest, since some algorithms can tolerate jumbled matrices on input, and sometimes the sort can be skipped entirely. However, if the matrix or vector will be immediately exported in unjumbled form, or provided as input to a method that requires it to not be jumbled, then sorting it during the matrix multiplication is faster. By default, these methods leave the result in jumbled form (a *lazy sort*), if `GxB_SORT` is set to zero (`GrB_DEFAULT`). A nonzero value will inform the matrix multiplication to sort its result, instead.

- `GxB_COMPRESSION` selects the compression method for serialization. The default is ZSTD (level 1). See Section 6.11 for other options.

The next sections describe the methods for a `GrB_Descriptor`:

GraphBLAS function	purpose	Section
`GrB_Descriptor_new`	create a descriptor	6.14.1
`GrB_Descriptor_wait`	wait for a descriptor	6.14.2
`GrB_get`	get a parameter from a descriptor	10.13
`GrB_set`	set a parameter in a descriptor	10.13
`GrB_Descriptor_free`	free a descriptor	6.14.3

6.14.1 GrB_Descriptor_new: create a new descriptor

```
GrB_Info GrB_Descriptor_new     // create a new descriptor
(
    GrB_Descriptor *descriptor  // handle of descriptor to create
) ;
```

`GrB_Descriptor_new` creates a new descriptor, with all fields set to their defaults (output is not replaced, the mask is not complemented, the mask is valued not structural, neither input matrix is transposed, the method used in `C=A*B` is selected automatically, and `GrB_mxm` leaves the final sort as pending work).

6.14.2 GrB_Descriptor_wait: wait for a descriptor

```
GrB_Info GrB_wait                   // wait for a descriptor
(
    GrB_Descriptor descriptor,      // descriptor to wait for
    int mode                        // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

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

6.14.3 GrB_Descriptor_free: free a descriptor

```
GrB_Info GrB_free               // free a descriptor
(
    GrB_Descriptor *descriptor  // handle of descriptor to free
) ;
```

`GrB_Descriptor_free` frees a descriptor. Either usage:

```
GrB_Descriptor_free (&descriptor) ;
GrB_free (&descriptor) ;
```

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

6.14.4 GrB_DESC_*: built-in descriptors

Built-in descriptors are listed in the table below. A dash in the table indicates the default. These descriptors may not be modified or freed. Attempts to modify them result in an error (`GrB_INVALID_VALUE`); attempts to free them are silently ignored.

Descriptor	`OUTP`	`MASK` structural	`MASK` complement	`INP0`	`INP1`
`GrB_NULL`	-	-	-	-	-
`GrB_DESC_T1`	-	-	-	-	`GrB_TRAN`
`GrB_DESC_T0`	-	-	-	`GrB_TRAN`	-
`GrB_DESC_T0T1`	-	-	-	`GrB_TRAN`	`GrB_TRAN`
`GrB_DESC_C`	-	-	`GrB_COMP`	-	-
`GrB_DESC_CT1`	-	-	`GrB_COMP`	-	`GrB_TRAN`
`GrB_DESC_CT0`	-	-	`GrB_COMP`	`GrB_TRAN`	-
`GrB_DESC_CT0T1`	-	-	`GrB_COMP`	`GrB_TRAN`	`GrB_TRAN`
`GrB_DESC_S`	-	`GrB_STRUCTURE`	-	-	-
`GrB_DESC_ST1`	-	`GrB_STRUCTURE`	-	-	`GrB_TRAN`
`GrB_DESC_ST0`	-	`GrB_STRUCTURE`	-	`GrB_TRAN`	-
`GrB_DESC_ST0T1`	-	`GrB_STRUCTURE`	-	`GrB_TRAN`	`GrB_TRAN`
`GrB_DESC_SC`	-	`GrB_STRUCTURE`	`GrB_COMP`	-	-
`GrB_DESC_SCT1`	-	`GrB_STRUCTURE`	`GrB_COMP`	-	`GrB_TRAN`
`GrB_DESC_SCT0`	-	`GrB_STRUCTURE`	`GrB_COMP`	`GrB_TRAN`	-
`GrB_DESC_SCT0T1`	-	`GrB_STRUCTURE`	`GrB_COMP`	`GrB_TRAN`	`GrB_TRAN`
`GrB_DESC_R`	`GrB_REPLACE`	-	-	-	-
`GrB_DESC_RT1`	`GrB_REPLACE`	-	-	-	`GrB_TRAN`
`GrB_DESC_RT0`	`GrB_REPLACE`	-	-	`GrB_TRAN`	-
`GrB_DESC_RT0T1`	`GrB_REPLACE`	-	-	`GrB_TRAN`	`GrB_TRAN`
`GrB_DESC_RC`	`GrB_REPLACE`	-	`GrB_COMP`	-	-
`GrB_DESC_RCT1`	`GrB_REPLACE`	-	`GrB_COMP`	-	`GrB_TRAN`
`GrB_DESC_RCT0`	`GrB_REPLACE`	-	`GrB_COMP`	`GrB_TRAN`	-
`GrB_DESC_RCT0T1`	`GrB_REPLACE`	-	`GrB_COMP`	`GrB_TRAN`	`GrB_TRAN`
`GrB_DESC_RS`	`GrB_REPLACE`	`GrB_STRUCTURE`	-	-	-
`GrB_DESC_RST1`	`GrB_REPLACE`	`GrB_STRUCTURE`	-	-	`GrB_TRAN`
`GrB_DESC_RST0`	`GrB_REPLACE`	`GrB_STRUCTURE`	-	`GrB_TRAN`	-
`GrB_DESC_RST0T1`	`GrB_REPLACE`	`GrB_STRUCTURE`	-	`GrB_TRAN`	`GrB_TRAN`
`GrB_DESC_RSC`	`GrB_REPLACE`	`GrB_STRUCTURE`	`GrB_COMP`	-	-
`GrB_DESC_RSCT1`	`GrB_REPLACE`	`GrB_STRUCTURE`	`GrB_COMP`	-	`GrB_TRAN`
`GrB_DESC_RSCT0`	`GrB_REPLACE`	`GrB_STRUCTURE`	`GrB_COMP`	`GrB_TRAN`	-
`GrB_DESC_RSCT0T1`	`GrB_REPLACE`	`GrB_STRUCTURE`	`GrB_COMP`	`GrB_TRAN`	`GrB_TRAN`

6.15 GrB_free: free any GraphBLAS object

Each of the ten objects has `GrB_*_new` and `GrB_*_free` methods that are specific to each object. They can also be accessed by a generic function, `GrB_free`, that works for all ten objects. If `G` is any of the ten objects, the statement

```
GrB_free (&G) ;
```

frees the object and sets the variable `G` to `NULL`. It is safe to pass in a `NULL` handle, or to free an object twice:

```
GrB_free (NULL) ;       // SuiteSparse:GraphBLAS safely does nothing
GrB_free (&G) ;         // the object G is freed and G set to NULL
GrB_free (&G) ;         // SuiteSparse:GraphBLAS safely does nothing
```

However, the following sequence of operations is not safe. The first two are valid but the last statement will lead to undefined behavior.

```
H = G ;                // valid; creates a 2nd handle of the same object
GrB_free (&G) ;        // valid; G is freed and set to NULL; H now undefined
GrB_some_method (H) ;  // not valid; H is undefined
```

Some objects are predefined, such as the built-in types. If a user application attempts to free a built-in object, SuiteSparse:GraphBLAS will safely do nothing. The `GrB_free` function in SuiteSparse:GraphBLAS always returns `GrB_SUCCESS`.

7 The mask, accumulator, and replace option

After a GraphBLAS operation computes a result $\mathbf{T}$, (for example, $\mathbf{T} = \mathbf{AB}$ for `GrB_mxm`), the results are assigned to an output matrix $\mathbf{C}$ via the mask/accumulator phase, written as $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$. This phase is affected by the `GrB_REPLACE` option in the descriptor, the presence of an optional binary accumulator operator ($\odot$), the presence of the optional mask matrix $\mathbf{M}$, and the status of the mask descriptor. The interplay of these options is summarized in Table 1.

The mask $\mathbf{M}$ may be present, or not. It may be structural or valued, and it may be complemented, or not. These options may be combined, for a total of 8 cases, although the structural/valued option as no effect if $\mathbf{M}$ is not present. If $\mathbf{M}$ is not present and not complemented, then m_{ij} is implicitly true. If not present yet complemented, then all m_{ij} entries are implicitly zero; in this case, $\mathbf{T}$ need not be computed at all. Either $\mathbf{C}$ is not modified, or all its entries are cleared if the replace option is enabled. If $\mathbf{M}$ is present, and the structural option is used, then m_{ij} is treated as true if it is an entry in the matrix (its value is ignored). Otherwise, the value of m_{ij} is used. In both cases, entries not present are implicitly zero. These values are negated if the mask is complemented. All of these various cases are combined to give a single effective value of the mask at position ij.

The combination of all these options are presented in the Table 1. The first column is the `GrB_REPLACE` option. The second column lists whether or not the accumulator operator is present. The third column lists whether or not c_{ij} exists on input to the mask/accumulator phase (a dash means that it does not exist). The fourth column lists whether or not the entry t_{ij} is present in the result matrix $\mathbf{T}$. The mask column is the final effective value of m_{ij}, after accounting for the presence of $\mathbf{M}$ and the mask options. Finally, the last column states the result of the mask/accum step; if no action is listed in this column, then c_{ij} is not modified.

Several important observations can be made from this table. First, if no mask is present (and the mask-complement descriptor option is not used), then only the first half of the table is used. In this case, the `GrB_REPLACE` option has no effect. The entire matrix $\mathbf{C}$ is modified.

Consider the cases when c_{ij} is present but t_{ij} is not, and there is no mask or the effective value of the mask is true for this ij position. With no accumulator operator, c_{ij} is deleted. If the accumulator operator is present and the replace option is not used, c_{ij} remains unchanged.

repl	accum	$\mathbf{C}$	$\mathbf{T}$	mask	action taken by $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$
-	-	c_{ij}	t_{ij}	1	$c_{ij} = t_{ij}$, update
-	-	-	t_{ij}	1	$c_{ij} = t_{ij}$, insert
-	-	c_{ij}	-	1	delete c_{ij} because t_{ij} not present
-	-	-	-	1	
-	-	c_{ij}	t_{ij}	0	
-	-	-	t_{ij}	0	
-	-	c_{ij}	-	0	
-	-	-	-	0	
yes	-	c_{ij}	t_{ij}	1	$c_{ij} = t_{ij}$, update
yes	-	-	t_{ij}	1	$c_{ij} = t_{ij}$, insert
yes	-	c_{ij}	-	1	delete c_{ij} because t_{ij} not present
yes	-	-	-	1	
yes	-	c_{ij}	t_{ij}	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	t_{ij}	0	
yes	-	c_{ij}	-	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	-	0	
-	yes	c_{ij}	t_{ij}	1	$c_{ij} = c_{ij} \odot t_{ij}$, apply accumulator
-	yes	-	t_{ij}	1	$c_{ij} = t_{ij}$, insert
-	yes	c_{ij}	-	1	
-	yes	-	-	1	
-	yes	c_{ij}	t_{ij}	0	
-	yes	-	t_{ij}	0	
-	yes	c_{ij}	-	0	
-	yes	-	-	0	
yes	yes	c_{ij}	t_{ij}	1	$c_{ij} = c_{ij} \odot t_{ij}$, apply accumulator
yes	yes	-	t_{ij}	1	$c_{ij} = t_{ij}$, insert
yes	yes	c_{ij}	-	1	
yes	yes	-	-	1	
yes	yes	c_{ij}	t_{ij}	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	t_{ij}	0	
yes	yes	c_{ij}	-	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	-	0	

Table 1: Results of the mask/accumulator phase.

When there is no mask and the mask GrB_COMP option is not selected, the table simplifies (Table 2). The GrB_REPLACE option no longer has any effect. The GrB_SECOND_T binary operator when used as the accumulator unifies the first cases, shown in Table 3. The only difference now is the behavior when c_{ij} is present but t_{ij} is not. Finally, the effect of GrB_FIRST_T as the accumulator is shown in Table 4.

accum	$\mathbf{C}$	$\mathbf{T}$	action taken by $\mathbf{C} = \mathbf{C} \odot \mathbf{T}$
-	c_{ij}	t_{ij}	$c_{ij} = t_{ij}$, update
-	-	t_{ij}	$c_{ij} = t_{ij}$, insert
-	c_{ij}	-	delete c_{ij} because t_{ij} not present
-	-	-	
yes	c_{ij}	t_{ij}	$c_{ij} = c_{ij} \odot t_{ij}$, apply accumulator
yes	-	t_{ij}	$c_{ij} = t_{ij}$, insert
yes	c_{ij}	-	
yes	-	-	

Table 2: When no mask is present (and not complemented).

accum	$\mathbf{C}$	$\mathbf{T}$	action taken by $\mathbf{C} = \mathbf{C} \odot \mathbf{T}$
yes	c_{ij}	t_{ij}	$c_{ij} = t_{ij}$, apply GrB_SECOND accumulator
yes	-	t_{ij}	$c_{ij} = t_{ij}$, insert
yes	c_{ij}	-	
yes	-	-	

Table 3: No mask, with the SECOND operator as the accumulator.

accum	$\mathbf{C}$	$\mathbf{T}$	action taken by $\mathbf{C} = \mathbf{C} \odot \mathbf{T}$
yes	c_{ij}	t_{ij}	
yes	-	t_{ij}	$c_{ij} = t_{ij}$, insert
yes	c_{ij}	-	
yes	-	-	

Table 4: No Mask, with the FIRST operator as the accumulator.

8 GxB_Context: controlling computational resources

SuiteSparse:GraphBLAS v8.0.0 adds a new object, the `GxB_Context`, which controls the number of threads used by OpenMP. In the future, this same object will control the number of GPUs used.

The `GxB_Context` object is not needed if the user application is itself single threaded, with all parallelism is inside GraphBLAS itself. The object is also not needed if the user application is multi-threaded, but all user threads create the same number of threads inside GraphBLAS (say each using a single thread). In that case, `GrB_set(GrB_GLOBAL,1,GxB_NTHREADS)` can be used (for example).

However, suppose the user application creates 5 threads of its own, on a machine with 16 cores, and each thread wants to use a different number of threads inside GraphBLAS (one user thread uses 8 OpenMP threads and the the other four use 2 each, for example). This is where the `GxB_Context` object becomes essential.

The default context is `GxB_CONTEXT_WORLD`, which is not created by the user application but it can be modified. If a user thread does not create its own context, then its computational resources are determine by this `GxB_CONTEXT_WORLD` object. The following `GrB_set/get` methods access this global object without naming it directly (where `chunk` is a `GrB_Scalar` of type `GrB_FP64` or `GrB_FP32`):

- `GrB_set (GrB_GLOBAL, nthreads, GxB_NTHREADS)`
- `GrB_get (GrB_GLOBAL, &nthreads, GxB_NTHREADS)`
- `GrB_set (GrB_GLOBAL, chunk, GxB_CHUNK)`
- `GrB_get (GrB_GLOBAL, chunk, GxB_CHUNK)`

The above methods control the OpenMP threads used by all user threads in the user application. To allow each user thread to control its own OpenMP threading, each user thread needs to create its own Context object via `GxB_Context_new`. Next, the user thread must *engage* this context via `GxB_Context_engage`; all subsequent calls to GraphBLAS from this particular user thread will then use the number of OpenMP threads dictated by this particular context.

Engaging a GxB_Context object assigns to a threadprivate space accessible only by this particular user thread, so that any calls to GraphBLAS can access the settings in this object.

The opposite operation is to *disengage* a context. This removes a particular object from the threadprivate space of the user thread that is disengaging its context.

After a context object is created, the user thread that owns it can modify its settings in this object. An example appears in the GraphBLAS/Demo folder, part of which is listed below.

```
#pragma omp parallel for num_threads (nouter) schedule (dynamic, 1)
for (int k = 0 ; k < nmat ; k++)
{
    // each user thread constructs its own context
    GxB_Context Context = NULL ;
    GxB_Context_new (&Context) ;
    GrB_set (Context, ninner, GxB_NTHREADS) ;
    GxB_Context_engage (Context) ;

    // kth user thread builds kth matrix with ninner threads
    GrB_Matrix A = NULL ;
    GrB_Matrix_new (&A, GrB_FP64, n, n) ;
    GrB_Matrix_build (A, I, J, X, nvals, GrB_PLUS_FP64) ;

    // free the matrix just built
    GrB_Matrix_free (&A) ;

    // each user thread frees its own context
    GxB_Context_disengage (Context) ;
    GxB_Context_free (&Context) ;
}
```

In this example, nouter user threads are created. Inside the parallel loop, each user thread creates and engages its own context object. In this simple example, each user thread then uses ninner threads to do some work, although in principle each user thread to request a different number of threads for each of its calls to GraphBLAS. This leads to nested parallelism, so to use this context object effectively, the nested parallelism feature of OpenMP must be enabled.

The next sections describe the methods for a GxB_Context:

GraphBLAS function	purpose	Section
`GxB_Context_new`	create a context	8.1
`GrB_get`	get a value from a context	10.14
`GrB_set`	set a value in a context	10.14
`GxB_Context_engage`	engage a context	8.2
`GxB_Context_disengage`	disengage a context	8.3
`GxB_Context_fprint`	check/print a context	13.11
`GxB_Context_free`	free a context	8.4
`GxB_Context_wait`	wait for a context	8.5

8.1 GxB_Context_new: create a new context

```
GrB_Info GxB_Context_new        // create a new context
(
    GxB_Context *Context        // handle of context to create
) ;
```

A new context is created and initialized with the current global settings for `GxB_NTHREADS` and `GxB_CHUNK`. See `GrB_get`. The context object will not have an effect on any calls to GraphBLAS until it is *engaged* by a user thread.

8.2 GxB_Context_engage: engaging context

```
GrB_Info GxB_Context_engage         // engage a Context
(
    GxB_Context Context             // Context to engage
) ;
```

GxB_Context_engage sets the provided Context object as the Context for this user thread. Multiple user threads can share a single Context. Any prior Context for this user thread is superseded by the new Context (the prior one is not freed). GrB_SUCCESS is returned, and future calls to GraphBLAS by this user thread will use the provided Context.

If the Context on input is the GxB_CONTEXT_WORLD object, then the current Context is disengaged. That is, the following calls have the same effect, setting the Context of this user thread to GxB_CONTEXT_WORLD:

```
GxB_Context_engage (GxB_CONTEXT_WORLD) ;
GxB_Context_disengage (NULL) ;
```

The result for both cases above is GrB_SUCCESS.

Error cases: If Context is NULL on input, GrB_NULL_POINTER is returned. If a non-NULL Context is provided but it is faulty in some way, then an error code is returned (GrB_INVALID_OBJECT or GrB_UNINITIALIZED_OBJECT). If an error code is returned, the current Context for this user thread is unmodified.

8.3 GxB_Context_disengage: disengaging context

```
GrB_Info GxB_Context_disengage      // disengage a Context
(
    GxB_Context Context             // Context to disengage
) ;
```

If a NULL Context is provided or if the Context input parameter is GxB_CONTEXT_WORLD, then any current Context for this user thread is disengaged. If a valid non-NULL Context is provided and it matches the current Context for this user thread, it is disengaged. In all of these cases, GrB_SUCCESS is returned. The user thread has no Context object and any subsequent calls to GraphBLAS functions will use the world Context, GxB_CONTEXT_WORLD.

Error cases: If a non-NULL Context is provided but it is faulty in some way, then an error code is returned (GrB_INVALID_OBJECT or GrB_UNINITIALIZED_OBJECT).

If a non-NULL Context is provided on input that doesn't match the current Context for this thread, then GrB_INVALID_VALUE is returned. If an error code is returned, the current Context for this user thread is unmodified.

8.4 GxB_Context_free: free a context

```
GrB_Info GrB_free           // free a context
(
    GxB_Context *Context    // handle of Context to free
) ;
```

GxB_Context_free frees a descriptor. Either usage:

```
GxB_Context_free (&Context) ;
GrB_free (&Context) ;
```

frees the Context and sets Context to NULL. It safely does nothing if passed a NULL handle, or if Context == NULL on input.

8.5 GxB_Context_wait: wait for a context

```
GrB_Info GrB_wait               // wait for a context
(
    GxB_Context Context,        // context to wait for
    int mode                    // GrB_COMPLETE or GrB_MATERIALIZE
) ;
```

After creating or modifying a context, a GraphBLAS library may choose to exploit non-blocking mode to delay its creation. Currently, SuiteSparse:GraphBLAS currently does nothing except to ensure that Context is valid.

9 The SuiteSparse:GraphBLAS JIT

SuiteSparse:GraphBLAS v8.0 adds a new JIT feature that greatly improves performance of user-defined types and operators, and improves the performance of built-in operators as well. The JIT can compile kernels that are specific to the matrix type and the operators that work on it. In version v7.4.4 and prior versions, user-defined types and operators were handled by *generic* kernels that used function pointers for each operator and for any typecasting required. Even built-in types and operators were sometimes handled by the generic kernels, if any typecasting was done, or if the specific operator, monoid, or semiring was disabled when GraphBLAS was compiled.

9.1 Using the JIT

Using the JIT in a user application is simple: by default, there is nothing to do. LAGraph v1.0.1 can use the JIT (and PreJIT) kernels without changing a single line of code.[1]

Currently, the JIT compiles kernels for the CPU only, but a CUDA JIT is in progress to exploit NVIDIA GPUs, in collaboration with Joe Eaton and Corey Nolet, with NVIDIA.

When GraphBLAS is compiled, the `cmake` build system creates a *cache* folder where it will keep any kernels created and compiled by the JIT (both source code and compiled libraries for each kernel). The default folder is `~/.SuiteSparse/GrB8.0.0` for SuiteSparse:GraphBLAS version v8.0.0, where the tilde refers to the user's home directory. The version numbers in the folder name are set automatically, so that a new version will ignore kernels compiled by an older version of GraphBLAS. If the `GRAPHBLAS_CACHE_PATH` environment variable is set when GraphBLAS is compiled, that variable defines the folder. If the user's home directory cannot be determined and the `GRAPHBLAS_CACHE_PATH` environment variable is not set, then JIT compilation is disabled and only PreJIT kernels can be used. The optional environment variable, `GRAPHBLAS_CACHE_PATH`, is also read by `GrB_init` when the user application runs. See Section 9.1.11 for a description of the valid characters that can appear in the cache path.

[1] LAGraph v1.0.1 does not work with GraphBLAS v8.0.0 but not because of the JIT. LAGraph must be updated because it uses two deprecated GxB_SelectOp operators; these are replaced with GrB_IndexUnaryOp operators in the dev branch of LAGraph, to appear in a future release of LAGraph.

When the user application starts, it can modify the location of the cache folder after calling `GrB_init`. It can also modify the C compiler and its flags, and can control when and how the JIT is used. These changes are made via `GrB_set`, and can be queried via `GrB_get`; refer to Section 10 for details, and the `GxB_JIT_*` settings:

field	value	description
`GxB_JIT_C_COMPILER_NAME`	`char *`	C compiler for JIT kernels
`GxB_JIT_C_COMPILER_FLAGS`	`char *`	flags for the C compiler
`GxB_JIT_C_LINKER_FLAGS`	`char *`	link flags for the C compiler
`GxB_JIT_C_LIBRARIES`	`char *`	libraries to link against (no cmake)
`GxB_JIT_C_CMAKE_LIBS`	`char *`	libraries to link against (with cmake)
`GxB_JIT_C_PREFACE`	`char *`	C code as preface to JIT kernels
`GxB_JIT_C_CONTROL`	see below	CPU JIT control
`GxB_JIT_USE_CMAKE`	see below	CPU JIT control
`GxB_JIT_ERROR_LOG`	`char *`	error log file
`GxB_JIT_CACHE_PATH`	`char *`	folder with compiled kernels

To control the JIT in the MATLAB `@GrB` interface, use the `GrB.jit` method. Refer to `help GrB.jit` for details.

Kernels compiled during one run of a user application are kept in the cache folder, so that when the user application runs again, the kernels do not have to be compiled. If the kernel relies on user-defined types and/or operators, a check is made the first time the compiled kernel is loaded. If the current definition of the user-defined type or operator does not exactly match the definition when the kernel was compiled, then the compiled kernel is discarded and recompiled. The stale kernel is overwritten with the new one, so there is no need to for the user to take any action to delete the stale kernel from the cache path. If the cache path is changed via `GrB_set`, compiled kernels in the old cache folder are not copied over. New ones are compiled instead.

9.1.1 GxB_JIT_C_CONTROL

The usage of the CPU JIT can be controlled via `GrB_get/set` using the `GxB_JIT_C_CONTROL` setting. If the JIT is enabled at compile time, the initial setting is `GxB_JIT_ON`. If the JIT is disabled at compile time (by setting the cmake variable `GRAPHBLAS_USE_JIT` to `OFF`), the initial setting is `GxB_JIT_RUN`, so that any PreJIT kernels can be run. This setting can be modified; for example to disable the JIT and clear all loaded JIT kernels from memory, use:

```
GrB_set (GrB_GLOBAL, GxB_JIT_OFF, GxB_JIT_C_CONTROL) ;
```

The above call to GrB_set does not clear any PreJIT kernels, however, since those are integral components of the single compiled GraphBLAS library and cannot be cleared (see Section 9.3). It also does not clear any compiled user functions, created by the JIT for GxB_*Op_new when the input function pointer is NULL.

The following settings are available for GxB_JIT_C_CONTROL. For examples on how to use it, see GraphBLAS/Demo/Program/gauss_demo.c.

```
typedef enum
{
    GxB_JIT_OFF = 0,    // do not use the JIT: free all JIT kernels if loaded
    GxB_JIT_PAUSE = 1,  // do not run JIT kernels but keep any loaded
    GxB_JIT_RUN = 2,    // run JIT kernels if already loaded; no load/compile
    GxB_JIT_LOAD = 3,   // able to load and run JIT kernels; may not compile
    GxB_JIT_ON = 4,     // full JIT: able to compile, load, and run
}
GxB_JIT_Control ;
```

If the JIT is disabled at compile time via setting the GRAPHBLAS_USE_JIT option OFF, PreJIT kernels are still available, and can be controlled via the GxB_JIT_OFF, GxB_JIT_PAUSE, or GxB_JIT_RUN settings listed above. If the application tries to set the control to GxB_JIT_LOAD or GxB_JIT_ON, the setting is changed to GxB_JIT_RUN instead. This is not an error condition. The resulting setting can be queried via GrB_get, if desired.

If your copy of GraphBLAS has many PreJIT kernels compiled into it, or uses many run-time JIT kernels, turning of the JIT with GxB_JIT_OFF can be costly. This setting clears the entire JIT hash table. Renabling the JIT and using it will require the JIT table to be repopulated, including a check of each PreJIT kernel the first time they are used. If you wish to temporarily disable the JIT, consider switching the JIT control to GxB_JIT_PAUSE and then back to GxB_JIT_RUN to reenable the JIT.

9.1.2 JIT error handling

The JIT control setting can be changed by GraphBLAS itself, based on following error conditions. These changes affect all kernels, not just the kernel causing the error. If any of these cases occur, the call to GraphBLAS returns GxB_JIT_ERROR, unless GraphBLAS runs out of memory, in which case it returns GrB_OUT_OF_MEMORY instead. If the JIT is disabled through any of these errors, it can be detected by GrB_get to read the GxB_JIT_C_CONTROL state.

- When a kernel is loaded that relies on user-defined types and/or operators, the definitions in the previously compiled kernel are checked against the current definitions. If they do not match, the old one is discarded, and a new kernel will be compiled. However, if the control is set to GxB_JIT_LOAD, no new kernels may be compiled. To avoid a continual reloading and checking of stale kernels, the control is changed from GxB_JIT_LOAD to GxB_JIT_RUN. To solve this problem, delete the compiled kernel with the stale definition, or enable the full JIT by setting the control to GxB_JIT_ON so that the kernel can recompiled with the current definitions.

- If a new kernel is to be compiled with the control set to GxB_JIT_ON but the source file cannot be created in the cache folder, or a compiler error occurs, further compilation is disabled. The control is changed from GxB_JIT_ON to GxB_JIT_LOAD. To solve this problem, make sure your application has write permission to the cache path and that any user-defined types and operators are defined properly so that no syntax error is detected by the compiler.

- If a kernel is loaded but the lookup of the kernel function itself in the compiled library fails, the control is changed to GxB_JIT_RUN to prevent this error from occuring again. To solve this problem, delete the corrupted compiled kernel from the cache folder. This case is unlikely to occur since no user action can trigger it. It could indicate a system problem with loading the kernel, or some kind of compiler error that allows the kernel to be compiled but not loaded.

- If an out-of-memory condition occurs in the JIT, the JIT control is set to GxB_JIT_PAUSE. To solve this, try freeing up memory, use a larger system, or solve smaller problems.

As a result of this automatic change in the JIT control setting, after the first JIT error is returned, subsequent calls to GraphBLAS will likely succeed. GraphBLAS will use a generic kernel instead. To re-enable the JIT for subsequent calls to GraphBLAS, the user application must reset the `GxB_JIT_C_CONTROL` back to `GxB_JIT_ON`.

In many use cases of GraphBLAS (such as LAGraph), a function will create a type or operator, use it, and then free it just before returning. It would be far to costly to clear the loaded kernel and reload it each time the LAGraph function is called, so any kernels that use this type or operator are kept loaded when the type or operator is freed. The typical case is that when the LAGraph function is called again, it will recreate the type or operator with the identical name and definition. The kernels that use these types or operators will still be loaded and can thus be used with no overhead.

However, if a user-defined type or operator is freed and then redefined with the same name but a different definition, any loaded kernels should be freed. This case is not detected by GraphBLAS since it would be far too costly to check each time a previously loaded kernel is called. As a result, this condition is only checked when the kernel is first loaded. To avoid this issue, if the user application frees a user-defined type or operator and creates a new one with a different definition but with the same name, clear all prior kernels by setting the control to `GxB_JIT_OFF`. Then turn the JIT back on with `GxB_JIT_ON`. This clears all run-time JIT kernels so that they will be checked when reloaded, and recompiled if their definitions changed. All PreJIT kernels are flagged as unchecked, just as they were flagged by `GrB_init`, so that they will be checked the next time they run.

9.1.3 GxB_JIT_C_COMPILER_NAME

The `GxB_JIT_C_COMPILER_NAME` string is the name of the C compiler to use, or its full path. By default it is set to the C compiler used to compile GraphBLAS itself.

9.1.4 GxB_JIT_C_COMPILER_FLAGS

The `GxB_JIT_C_COMPILER_FLAGS` string is the C compiler flags. By default it is set to the C compiler flags used to compile GraphBLAS itself.

9.1.5 GxB_JIT_C_LINKER_FLAGS

The `GxB_JIT_C_LINKER_FLAGS` string only affects the kernel compilation when cmake is not used to compile the kernels (see Section 9.1.9). By default it is set to the C link flags used to compile GraphBLAS itself. If cmake is used to compile the kernels, then it determines the linker flags itself, and this cannot be modified.

9.1.6 GxB_JIT_C_LIBRARIES

The `GxB_JIT_C_LIBRARIES` string is used to set the libraries to link against when cmake is not being used to compile the kernels (see Section 9.1.9). For example, on Linux it is set by default to the `-lm`, `-ld`, and OpenMP libraries used to link GraphBLAS itself. Any standalone library name is prepended with `-l`. If cmake is used to compile the kernels, this string is ignored.

9.1.7 GxB_JIT_C_CMAKE_LIBS

The `GxB_JIT_C_LIBRARIES` string is used to set the libraries to link against when cmake is being used to compile the kernels (see Section 9.1.9). For example, on Linux it is set by default to the `m`, `dl`, and OpenMP libraries used to link GraphBLAS itself. Libraries in the string should normally be separated by semicolons. If cmake is not used to compile the kernels, this string is ignored.

9.1.8 GxB_JIT_C_PREFACE

The `GxB_JIT_C_PREFACE` string is added at the top of each JIT kernel. It is useful for providing additional `#include` files that GraphBLAS does not provide. It can also be useful for diagnostics and for configuring the `PreJIT`. For example, suppose you wish to tag specific kernels as having been constructed for particular parts of an application. The application can modify this string to some unique comment, and then run some benchmarks that call GraphBLAS. Any JIT kernels created will be tagged with this unique comment, which may be helpful to select specific kernels to copy into the `PreJIT` folder.

9.1.9 GxB_JIT_USE_CMAKE

Two methods are provided for compiling the JIT kernels: cmake, and a direct compiler/link command. On Windows, only cmake may be used, and this setting is ignored (it is always true). On Linux or Mac, the default is false since a direct compile/link is faster. However, it is possible that some compilers are not handled properly with this method, so cmake can also be used on those platforms by setting the value of `GxB_JIT_USE_CMAKE` to true.

Normally the same version of cmake should be used to compile both GraphBLAS and the JIT kernels. However, compiling GraphBLAS itself requires cmake v3.16 or later (v3.19 for some options), while compiling the JIT kernels only requires cmake v3.13 or later.

9.1.10 GxB_JIT_ERROR_LOG

The `GxB_JIT_ERROR_LOG` string is the filename of the optional error log file. By default, this string is empty, which means that any compiler errors are routed to the `stderr` output of the user process. If set to a non-empty string, any compiler errors are appended to this file. The string may be `NULL`, which means the same as an empty string.

9.1.11 GxB_JIT_CACHE_PATH

The `GxB_JIT_CACHE_PATH` string is the full path to the user's cache folder (described above). The default on Linux/Mac is `~/.SuiteSparse/GrB8.0.0` for GraphBLAS version 8.0.0. On Windows, the cache folder is created inside the user's `LOCALAPPDATA` folder, called `SuiteSparse/GrB8.0.0`. When GraphBLAS starts, `GrB_init` checks if the `GRAPHBLAS_CACHE_PATH` environment variable exists, and initializes the cache path with that value instead of using the default.

Restrictions: the cache path is sanitized for security reasons. No spaces are permitted. Backslashes are converted into forward slashes. It can contain only charactors in the following list:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789.-_/
```

In addition, the second character in the string is allowed to be the colon character (`:`) to allow for the use of Windows drive letters. Any character outside of these rules is converted into an underscore (`_`).

9.2 Compilation options: GRAPHBLAS_USE_JIT and GRAPHBLAS_COMPACT

The CPU JIT can be disabled at compile time by setting the `GRAPHBLAS_USE_JIT` option `OFF` in the cmake build options. Good performance will be obtained only by using the `FactoryKernels` or the `PreJIT` kernels that are compiled into GraphBLAS when it is first compiled with `cmake`. By default, `GRAPHBLAS_USE_JIT` is `ON`, to enable the CPU JIT.

With the introduction of the JIT kernels, it is now possible to obtain good performance in GraphBLAS without compiling the many *factory kernels* that appear in the `GraphBLAS/Source/FactoryKernels` directory. If the JIT is enabled, GraphBLAS will still be fast, once the JIT kernels are compiled, or by using any `PreJIT` kernels. To compile GraphBLAS without its `FactoryKernels`, enable the `COMPACT` option in the cmake build options. By default, `COMPACT` is off, to enable the `FactoryKernels`.

When GraphBLAS is compiled with `GRAPHBLAS_USE_JIT` set to `OFF`, the `GxB_JIT_C_CONTROL` may be set to `GxB_JIT_OFF`, `GxB_JIT_PAUSE`, or `GxB_JIT_RUN`. No kernels will be loaded at run-time (the `GxB_JIT_LOAD` setting is disabled and treated as `GxB_JIT_RUN`), and no new kernels will be compiled at run-time (the `GxB_JIT_ON` is disabled and treated as `GxB_JIT_RUN`). Only pre-existing `PreJIT` kernels can be run, described in Section 9.3.

If both `GRAPHBLAS_USE_JIT` is set `OFF` and `GRAPHBLAS_COMPACT` is set `ON`, all features of GraphBLAS will be functional. The only fast kernels available will be the `PreJIT` kernels (if any). Otherwise, generic kernels will be used, in which every single operator is implemented with a function pointer, and every scalar assignment requires a `memcpy`. Generic kernels are slow, so using this combination of options is not recommended when preparing GraphBLAS for production use, benchmarking, or for a Linux distro or other widely-used distribution, unless you are able to run your application in advance and create all the JIT kernels you need, and then copy them into `GraphBLAS/PreJIT`. This would be impossible to do for a general-purpose case such as a Linux distro, but feasible for a more targetted application such as FalkorDB.

9.3 Adding PreJIT kernels to GraphBLAS

When GraphBLAS runs, it constructs JIT kernels in the user's cache folder, which by default is `~/.SuiteSparse/GrB8.0.0` for v8.0.0. The kernels placed in a subfolder (`c`) and inside that folder they are further subdivided arbitrarily into subfolders (via an arbitary hash). The files are split into subfolders because a single folder may grow too large for efficient access. Once GraphBLAS has generated some kernels, some or all of them kernels can then incorporated into the compiled GraphBLAS library by copying them into the `GraphBLAS/PreJIT` folder. Be sure to move any `*.c` files into the single `GraphBLAS/PreJIT` folder; do not keep the subfolder structure.

If GraphBLAS is then recompiled via cmake, the build system will detect these kernels, compile them, and make them available as pre-compiled JIT kernels. The kernels are no longer "Just-In-Time" kernels since they are not compiled at run-time. They are refered to as `PreJIT` kernels since they were at one time created at run-time by the GraphBLAS JIT, but are now compiled into GraphBLAS before it runs.

It's that simple. Just copy the source files for any kernels you want from your cache folder (typically `~/.SuiteSparse/GrB8.0.0/c`) into `GraphBLAS/PreJIT`, and recompile GraphBLAS. There's no need to change any other cmake setting, and no need to do anything different in any applications that use GraphBLAS. Do not copy the compiled libraries; they are not needed and will be ignored. Just copy the `*.c` files.

If the resulting GraphBLAS library is installed for system-wide usage (say in a Linux distro, Python, RedisGraph, etc), the `GraphBLAS/PreJIT` kernels will be available to all users of that library. They are not disabled by the `GRAPHBLAS_USE_JIT` option.

Once these kernels are moved to `GraphBLAS/PreJIT` and GraphBLAS is recompiled, they can be deleted from the cache folder. However, even if they are left there, they will not be used since GraphBLAS will find these kernels as PreJIT kernels inside the compiled library itself (`libgraphblas.so` on Linux, `libgraphblas.dylib` on the Mac). GraphBLAS will not be any slower if these kernels are left in the cache folder, and the compiled library size will not be affected.

If the GraphBLAS version is changed at all (even in the last digit), all `GB_jit_*.c` files in the `GraphBLAS/PreJIT` folder should be deleted. The version mismatch will be detected during the call to `GrB_init`, and any stale kernels will be safely ignored. Likewise, if a user-defined type or operator is changed, the relevant kernels should also be deleted from `GraphBLAS/PreJIT`. For example, the `GraphBLAS/Demo/Program/gauss_demo.c` program creates a user-defined `gauss` type, and two operators, `addgauss` and `multgauss`. It then intentionally changes one of the operators just to test this feature. If the type and/or operators are changed, then the `*gauss*.c` files in the

GraphBLAS/PreJIT folder should be deleted.

GraphBLAS will safely detect any stale PreJIT kernels by checking them the first time they are run after calling GrB_init and will not use them if they are found to be stale. If the JIT control is set to GxB_JIT_OFF all PreJIT kernels are flagged as unchecked. If the JIT is then renabled by setting the control to GxB_JIT_RUN or GxB_JIT_ON, all PreJIT kernels will be checked again and any stale kernels will be detected.

If a stale PreJIT kernel is found, GraphBLAS will use its run-time JIT to compile new ones with the current definitions, or it will punt to a generic kernel if JIT compilation is disabled. GraphBLAS will be functional, and fast if it can rely on a JIT kernel, but the unusable stale PreJIT kernels take up space inside the compiled GraphBLAS library. The best practice is to delete any stale kernels from the GraphBLAS/PreJIT folder, or replace them with newly compiled JIT kernels from the cache folder, and recompile GraphBLAS.

It is safe to copy only a subset of the JIT kernels from the cache folder into GraphBLAS/PreJIT. You may also delete any files in GraphBLAS/PreJIT and recompile GraphBLAS without those kernels. If GraphBLAS encounters a need for a particular kernel that has been removed from GraphBLAS/PreJIT, it will create it at run-time via the JIT, if permitted. If not permitted, by either compiling GraphBLAS with the GRAPHBLAS_USE_JIT option set ot OFF, or by using GxB_JIT_C_CONTROL at run-time, the factory kernel or generic kernel will be used instead. The generic kernel will be slower than the PreJIT or JIT kernel, but GraphBLAS will still be functional.

In addition to a single README.txt file, the GraphBLAS/PreJIT folder includes a .gitignore file that prevents any files in the folder from being synced via git. If you wish to add your PreJIT kernels to a fork of GraphBLAS, you will need to revise this .gitignore file.

9.4 JIT and PreJIT performance considerations

To create a good set of PreJIT kernels for a particular user application, it is necessary to run the application with many different kinds of workloads. Each JIT or PreJIT kernel is specialized to the particular matrix format, data type, operators, and descriptors of its inputs. GraphBLAS can change a matrix format (from sparse to hypersparse, for example), at its discretion, thus triggering the use of a different kernel. Some GraphBLAS methods use heuristics to select between different methods based upon the sparsity structure or estimates of the kind or amount of work required. In these cases, entirely different kernels will be compiled. As a result, it's very difficult to predict which kernels GraphBLAS will find the need to compile, and thus a wide set of test cases should be used in an application to allow GraphBLAS to generate as many kernels as could be expected to appear in production use.

GraphBLAS can encounter very small matrices, and it will often select its bitmap format to store them. This change of format will trigger a different kernel than the sparse or hypersparse cases. There are many other cases like that where specific kernels are only needed for small problems. In this case, compiling an entirely new kernel is costly, since using a compiled kernel will be no faster than the generic kernel. When benchmarking an application to allow GraphBLAS to compile its JIT kernels, it may be useful to pause the JIT via `GxB_JIT_PAUSE`, `GxB_JIT_RUN`, or `GxB_JIT_LOAD`, when the application knows it is calling GraphBLAS for tiny problems. These three settings keep any loaded JIT kernels in memory, but pauses the compilation of any new JIT kernels. Then the control can be reset to `GxB_JIT_ON` once the application finishes with its tiny problems and moves to larger ones where the JIT will improve performance. A future version of GraphBLAS may allow this heuristic to be implemented inside GraphBLAS itself, but for now, the JIT does not second guess the user application; if it wants a new kernel, the JIT will compile it if the control is set to `GxB_JIT_ON`.

9.5 Mixing JIT kernels: MATLAB and Apple Silicon

In general, the JIT kernels compiled by the C interface and the kernels compiled while using GraphBLAS in MATLAB are interchangable, and the same cache folder can be used for both. This is the default.

However, when using the `@GrB` MATLAB interface to GraphBLAS on Apple Silicon, the MATLAB JIT kernels are compiled as x86 binaries and executed inside MATLAB via Rosetta. The pure C installation may compile native Arm64 binaries for its JIT kernels. Do not mix the two. In this case, set another cache path for MATLAB using `GrB.jit` in MATLAB, or using `GrB_set` in the C interface for your native Arm64 binaries.

9.6 Updating the JIT when GraphBLAS source code changes

If you edit the GraphBLAS source code itself or add any files to `GraphBLAS/PreJIT`, read the instructions in `GraphBLAS/JITpackage/README.txt` for details on how to update the JIT source code.

If your cache folder (`~/.SuiteSparse/GrBx.y.z`) changes in any way except via GraphBLAS itself, simply delete your cache folder. GraphBLAS will then reconstruct the kernels there as needed.

9.7 Future plans for the JIT and PreJIT

9.7.1 Kernel fusion

The introduction of the JIT and its related PreJIT kernels allow for the future exploitation of kernel fusion via an aggressive exploitation of the GraphBLAS non-blocking mode. In that mode, multiple calls to GraphBLAS can be fused into a single kernel. There are far to many possible variants to allow a fused kernel to appear in the `GraphBLAS/Source/FactoryKernels` folder, but specific fused kernels could be created by the JIT.

9.7.2 Heuristics for controlling the JIT

As mentioned in Section 9.4, GraphBLAS may compile JIT kernels that are used for only tiny problems where the compile time of a single kernel will dominate any performance gains from using the compiled kernel. A heuristic could be introduced so that it compiles them only for larger problems. The possible downside of this approach is that the same JIT kernels might be needed later for larger problems.

9.7.3 CUDA / SYCL / OpenCL kernels

The CUDA JIT will enable NVIDIA GPUs to be exploited. There are simply too many kernels to create at compile time as the "factory kernels." This CUDA JIT is in progress. A related JIT for SYCL / OpenCL kernels is under consideration.

9.7.4 Better performance for multithreaded user programs:

This version is thread-safe when used in a multithread user application, but a better JIT critical section (many readers, one writer) might be needed. The current critical section may be sufficiently fast since the typical case of work done inside the critical section is a single hash table lookup. However, the performance issues related to this have not been tested. This has no effect if all parallelism is exploited only within GraphBLAS. It only affects the case when multiple user threads each call GraphBLAS in parallel (using the `GxB_Context`; see Section 8).

10 GraphBLAS Options (GrB_get and GrB_set)

The latest v2.1 C API adds a pair of methods, `GrB_get` and `GrB_set` that allows the user to query GraphBLAS objects and change their state. These two methods are polymorphic wrappers for a suite of methods for each object. They replace the `GxB_get` and `GxB_set` methods in SuiteSparse:GraphBLAS v8.* and earlier.

The general polymorphic signatures of these methods are given below:

```
GrB_Info GrB_get (object, value, int field) ;
GrB_Info GrB_set (object, value, int field) ;
GrB_Info GrB_set (object, void *value, int field, size_t s) ;
```

where `object` can be any GraphBLAS object. The `value` can be a `GrB_Scalar`, an `int32_t` (or a pointer to `int32_t` for `GrB_get`), a string (`char *`), or a `void *` pointer. In the latter case, `GrB_set` requires an additional parameter (`size_t s`) that specifies the size of the object pointed to by `void *value`.

The non-polymorphic names have the following format, where `[OBJ]` is any GraphBLAS object.

```
GrB_Info GrB_[OBJ]_get_[KIND] (object, value, int field) ;
GrB_Info GrB_[OBJ]_set_[KIND] (object, value, int field) ;
GrB_Info GrB_[OBJ]_set_VOID (object, void *value, int field, size_t s) ;
```

The `[KIND]` suffix defines the type of second parameter, and can be: `Scalar` (for `GrB_Scalar`), `String` (for `char *`), `INT32` (for `int32_t`), and `SIZE` (for `size_t *` for `GrB_get` only), and `VOID` (for `void *`).

The tables below list all the valid fields that can be used for each object. Each table contains four columns: (1) the `int`, (2) a column labelled R/W, (3) a column defining the C type, and a description. For the R/W column:

- If the R/W column of a table is R, then the value can be read by `GrB_get` but not written by `GrB_set`.
- R/W means that both `GrB_get` and `GrB_set` can be used any number of times, with different values.
- R/W1 means that `GrB_get` can be done multiple times, but `GrB_set` can be used only once. Subsequent calls to `GrB_set` return the error code `GrB_ALREADY_SET`.

- W means that only GrB_set can be used (any number of times), but GrB_get cannot be done.

The second parameter (value) of GrB_get and GrB_set can take on several different C types, and it can also be a GrB_Scalar that holds a value with the given C type (or that can be typecasted to the given C type):

- int32_t: For GrB_set the value parameter is int32_t. For GrB_get the value parameter is int32_t *. The following example sets the global number of threads, and then retrieves that value into nthreads.

  ```
  GrB_set (GrB_GLOBAL, 8, GxB_NTHREADS) ;
  int32_t nthreads ;
  GrB_get (GrB_GLOBAL, &nthreads, GxB_NTHREADS) ;
  printf ("nthreads: %d\n", nthreads) ;
  ```

  A GrB_Scalar can also be used for an int32_t value. For GrB_set, the scalar must not be empty. Here is the same example but using a GrB_Scalar instead:

  ```
  GrB_Scalar s ;
  GrB_Scalar_new (s, GrB_INT32) ;
  GrB_Scalar_setElement (s, 8) ;
  GrB_set (GrB_GLOBAL, s, GxB_NTHREADS) ;
  GrB_get (GrB_GLOBAL, s, GxB_NTHREADS) ;
  int32_t nthreads ;
  GrB_Scalar_extractElement (&nthreads, s) ;
  printf ("nthreads: %d\n", nthreads) ;
  ```

- char *: The value parameter is char * for both GrB_get and GrB_set. A GrB_Scalar cannot be used. The size of the string required for GrB_get is given by using a size_t * parameter with the same field. For example:

  ```
  size_t len ;
  GrB_get (GrB_GLOBAL, &len, GrB_NAME) ;
  char *name = malloc (len) ;
  GrB_get (GrB_GLOBAL, name, GrB_NAME) ;
  printf ("The library is: %s\n", name) ;
  free (name) ;
  ```

To get the current JIT C compiler and then set it to something else:

```
size_t len ;
GrB_get (GrB_GLOBAL, &len, GxB_JIT_C_COMPILER_NAME) ;
char *compiler = malloc (len) ;
GrB_get (GrB_GLOBAL, compiler, GxB_JIT_C_COMPILER_NAME) ;
printf ("The current JIT compiler is: %s\n", compiler) ;
GrB_set (GrB_GLOBAL, "gcc", GxB_JIT_C_COMPILER_NAME) ;
GrB_get (GrB_GLOBAL, &len, GxB_JIT_C_COMPILER_NAME) ;
char *new_compiler = malloc (len) ;
GrB_get (GrB_GLOBAL, new_compiler, GxB_JIT_C_COMPILER_NAME) ;
printf ("The new JIT compiler is: %s\n", new_compiler) ;
free (compiler) ;
free (new_compiler) ;
```

- Other scalar data types (typically `double`): Only a `GrB_Scalar` can be used. When using `GrB_set` with a `GrB_Scalar`, the scalar cannot be empty. For example, to get then set the global `GxB_HYPER_SWITCH` parameter to 0.3:

```
GrB_Scalar s ;
GrB_Scalar_new (s, GrB_FP64) ;
GrB_get (GrB_GLOBAL, s, GxB_HYPER_SWITCH) ;
double hs ;
GrB_Scalar_extractElement (&hs, s) ;
printf ("current hyper_switch: %g\n", hs) ;
GrB_Scalar_setElement (s, 0.3) ;
GrB_set (GrB_GLOBAL, s, GxB_HYPER_SWITCH) ;
```

- `void *`: This type is used for all other cases. For `GrB_get`, the array must have the right size, just like a `char *` string. Use the same field first, but with `size_t *value` as the second parameter to obtain the size of the `void *` array, then use `GrB_get` with a `void *` array of the right size. In some cases, the size is always the same. For example, to query the operator of a monoid:

```
GrB_BinaryOp op ;
GrB_get (GrB_PLUS_MONOID_FP64, (void *) &op, GxB_MONOID_OPERATOR) ;
assert (op == GrB_PLUS_FP64) ;
```

For `GrB_set`, a fourth parameter is required to tell GraphBLAS the size of the input array.

10.1 Enum types for get/set: GrB_Field, GrB_Orientation, and GrB_Type_Code

The get/set methods share a `int` (enum) type to specify which component of the object is to be set or retrieved. fields.

```
typedef enum
{
    // GrB_Descriptor only:
    GrB_OUTP_FIELD = 0,     // descriptor for output of a method
    GrB_MASK_FIELD = 1,     // descriptor for the mask input of a method
    GrB_INP0_FIELD = 2,     // descriptor for the first input of a method
    GrB_INP1_FIELD = 3,     // descriptor for the second input of a method

    // all objects, including GrB_GLOBAL:
    GrB_NAME = 10,          // name of the object, as a string

    // GrB_GLOBAL only:
    GrB_LIBRARY_VER_MAJOR = 11,     // SuiteSparse:GraphBLAS version
    GrB_LIBRARY_VER_MINOR = 12,
    GrB_LIBRARY_VER_PATCH = 13,
    GrB_API_VER_MAJOR = 14,         // C API version
    GrB_API_VER_MINOR = 15,
    GrB_API_VER_PATCH = 16,
    GrB_BLOCKING_MODE = 17,         // GrB_Mode

    // GrB_GLOBAL, GrB_Matrix, GrB_Vector, GrB_Scalar (and void * serialize?):
    GrB_STORAGE_ORIENTATION_HINT = 100, // GrB_Orientation

    // GrB_Matrix, GrB_Vector, GrB_Scalar (and void * serialize):
    GrB_EL_TYPE_CODE = 102,         // a GrB_Type_code (see below)
    GrB_EL_TYPE_STRING = 106,       // name of the type

    // GrB_*Op, GrB_Monoid, and GrB_Semiring:
    GrB_INP1_TYPE_CODE = 103,       // GrB_Type_code
    GrB_INP2_TYPE_CODE = 104,
    GrB_OUTP_TYPE_CODE = 105,
    GrB_INP1_TYPE_STRING = 107,     // name of the type, as a string
    GrB_INP2_TYPE_STRING = 108,
    GrB_OUTP_TYPE_STRING = 109,

    // GrB_Type (readable only):
    GrB_SIZE = 110,                 // size of the type

    // GrB_Type, GrB_UnaryOp, GrB_BinaryOp, GrB_IndexUnaryOp,
```

```
    // and GxB_IndexBinaryOp
    GxB_JIT_C_NAME = 7041,          // C type or function name
    GxB_JIT_C_DEFINITION = 7042,    // C typedef or function definition

    // GrB_Monoid and GrB_Semiring:
    GxB_MONOID_IDENTITY = 7043,     // monoid identity value
    GxB_MONOID_TERMINAL = 7044,     // monoid terminal value
    GxB_MONOID_OPERATOR = 7045,     // monoid binary operator

    // GrB_Semiring only:
    GxB_SEMIRING_MONOID = 7046,     // semiring monoid
    GxB_SEMIRING_MULTIPLY = 7047,   // semiring multiplicative op

    // GrB_BinaryOp and GxB_IndexBinaryOp:
    GxB_THETA_TYPE_CODE = 7050,     // for binary and index binary ops
    GxB_THETA_TYPE_STRING = 7051,

    // GrB_BinaryOp or GrB_Semiring:
    GxB_THETA = 7052,               // to get the value of theta

#define GxB_NTHREADS 7086
#define GxB_CHUNK 7087

    // GrB_get/GrB_set for GrB_Matrix and GrB_GLOBAL:
    GxB_HYPER_SWITCH = 7000,    // switch to hypersparse (double value)
    GxB_HYPER_HASH = 7048,      // hyper_hash control (int64 value)
    GxB_BITMAP_SWITCH = 7001,   // switch to bitmap (double value)

    // GrB_get for GrB_GLOBAL:
    GxB_LIBRARY_DATE = 7006,         // date of the library (char *)
    GxB_LIBRARY_ABOUT = 7007,        // about the library (char *)
    GxB_LIBRARY_URL = 7008,          // URL for the library (char *)
    GxB_LIBRARY_LICENSE = 7009,      // license of the library (char *)
    GxB_LIBRARY_COMPILE_DATE = 7010, // date library was compiled (char *)
    GxB_LIBRARY_COMPILE_TIME = 7011, // time library was compiled (char *)
    GxB_API_DATE = 7013,             // date of the API (char *)
    GxB_API_ABOUT = 7014,            // about the API (char *)
    GxB_API_URL = 7015,              // URL for the API (char *)
    GxB_COMPILER_VERSION = 7016,     // compiler version (3 int's)
    GxB_COMPILER_NAME = 7017,        // compiler name (char *)
    GxB_LIBRARY_OPENMP = 7018,       // library compiled with OpenMP
    GxB_MALLOC_FUNCTION = 7037,      // malloc function pointer
    GxB_CALLOC_FUNCTION = 7038,      // calloc function pointer
    GxB_REALLOC_FUNCTION = 7039,     // realloc function pointer
    GxB_FREE_FUNCTION = 7040,        // free function pointer
```

```
    // GrB_get/GrB_set for GrB_GLOBAL:
    GxB_GLOBAL_NTHREADS = GxB_NTHREADS,  // max number of threads to use
    GxB_GLOBAL_CHUNK = GxB_CHUNK,        // chunk size for small problems.
    GxB_BURBLE = 7019,               // diagnostic output (bool *)
    GxB_PRINTF = 7020,               // printf function diagnostic output
    GxB_FLUSH = 7021,                // flush function diagnostic output
    GxB_PRINT_1BASED = 7023,         // print matrices as 0-based or 1-based
    GxB_JIT_C_COMPILER_NAME = 7024,  // CPU JIT C compiler name
    GxB_JIT_C_COMPILER_FLAGS = 7025, // CPU JIT C compiler flags
    GxB_JIT_C_LINKER_FLAGS = 7026,   // CPU JIT C linker flags
    GxB_JIT_C_LIBRARIES = 7027,      // CPU JIT C libraries
    GxB_JIT_C_PREFACE = 7028,        // CPU JIT C preface
    GxB_JIT_C_CONTROL = 7029,        // CPU JIT C control
    GxB_JIT_CACHE_PATH = 7030,       // CPU/CUDA JIT path for compiled kernels
    GxB_JIT_C_CMAKE_LIBS = 7031,     // CPU JIT C libraries when using cmake
    GxB_JIT_USE_CMAKE = 7032,        // CPU JIT: use cmake or direct compile
    GxB_JIT_ERROR_LOG = 7033,        // CPU JIT: error log file

    // GrB_get for GrB_Matrix:
    GxB_SPARSITY_STATUS = 7034,     // hyper, sparse, bitmap or full (1,2,4,8)

    // GrB_get/GrB_set for GrB_Matrix:
    GxB_SPARSITY_CONTROL = 7036,    // sparsity control: 0 to 15; see below

} GxB_Option_Field ;

typedef enum
{
    GrB_ROWMAJOR = 0,
    GrB_COLMAJOR = 1,
    GrB_BOTH     = 2,
    GrB_UNKNOWN  = 3,
}
GrB_Orientation ;

typedef enum
{
    GrB_UDT_CODE    = 0,        // user-defined type
    GrB_BOOL_CODE   = 1,        // GraphBLAS: GrB_BOOL      C: bool
    GrB_INT8_CODE   = 2,        // GraphBLAS: GrB_INT8      C: int8_t
    GrB_UINT8_CODE  = 3,        // GraphBLAS: GrB_UINT8     C: uint8_t
    GrB_INT16_CODE  = 4,        // GraphBLAS: GrB_INT16     C: int16_t
    GrB_UINT16_CODE = 5,        // GraphBLAS: GrB_UINT16    C: uint16_t
    GrB_INT32_CODE  = 6,        // GraphBLAS: GrB_INT32     C: int32_t
```

```
    GrB_UINT32_CODE = 7,        // GraphBLAS: GrB_UINT32    C: uint32_t
    GrB_INT64_CODE  = 8,        // GraphBLAS: GrB_INT64     C: int64_t
    GrB_UINT64_CODE = 9,        // GraphBLAS: GrB_UINT64    C: uint64_t
    GrB_FP32_CODE   = 10,       // GraphBLAS: GrB_FP32      C: float
    GrB_FP64_CODE   = 11,       // GraphBLAS: GrB_FP64      C: double
    GxB_FC32_CODE   = 7070,     // GraphBLAS: GxB_FC32      C: float complex
    GxB_FC64_CODE   = 7071,     // GraphBLAS: GxB_FC64      C: double complex
}
GrB_Type_Code ;
```

10.2 Global Options (GrB_Global)

A single object GrB_GLOBAL whose type is GrB_Global is used to denote global settings to read or modify. To use it with GrB_get and GrB_set, pass in GrB_GLOBAL as the first parameter.

```
GrB_Info GrB_get (GrB_Global g, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_Global g, char *     value, int f) ;
GrB_Info GrB_get (GrB_Global g, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_Global g, size_t *   value, int f) ;
GrB_Info GrB_get (GrB_Global g, void *     value, int f) ;

GrB_Info GrB_set (GrB_Global g, GrB_Scalar value, int f) ;
GrB_Info GrB_set (GrB_Global g, char *     value, int f) ;
GrB_Info GrB_set (GrB_Global g, int32_t    value, int f) ;
GrB_Info GrB_set (GrB_Global g, void *     value, int f, size_t s) ;
```

int field	R/W	C type	description
GrB_LIBRARY_VER_MAJOR	R	int32_t	major version of the library
GrB_LIBRARY_VER_MINOR	R	int32_t	minor version of the library
GrB_LIBRARY_VER_PATCH	R	int32_t	patch version of the library
GrB_API_VER_MAJOR	R	int32_t	major version of the API
GrB_API_VER_MINOR	R	int32_t	major version of the API
GrB_API_VER_PATCH	R	int32_t	major version of the API
GrB_BLOCKING_MODE	R	int32_t	blocking mode (GrB_BLOCKING or GrB_NONBLOCKING)
GxB_LIBRARY_OPENMP	R	int32_t	if OpenMP is in use (true/false)

`int field`	R/W	C type	description
`GrB_STORAGE_ORIENTATION_HINT`	R/W	`int32_t`	see `GrB_Orientation`: default format for matrices.
`GxB_NTHREADS`	R/W	`int32_t`	number of OpenMP threads used. See Section 10.2.2.
`GxB_BURBLE`	R/W	`int32_t`	diagnostic output (true/false). See Section 10.2.1.
`GxB_PRINT_1BASED`	R/W	`int32_t`	matrices printed as 1-based or 0-based
`GxB_JIT_C_CONTROL`	R/W	`int32_t`	see Section 9
`GxB_JIT_USE_CMAKE`	R/W	`int32_t`	”
`GxB_HYPER_SWITCH`	R/W	`double`	global hypersparsity control. See Section 10.10.2.
`GxB_HYPER_HASH`	R/W	`int64_t`	global hypersparsity (hyper-hash) control
`GxB_CHUNK`	R/W	`double`	global chunk size for parallel task creation. See Section 10.2.2.
`GrB_NAME`	R	`char *`	name of the library (`"SuiteSparse:GraphBLAS"`)
`GxB_LIBRARY_DATE`	R	`char *`	library release date
`GxB_LIBRARY_ABOUT`	R	`char *`	details about the library
`GxB_LIBRARY_LICENSE`	R	`char *`	license of the library
`GxB_LIBRARY_COMPILE_DATE`	R	`char *`	date the library was compiled
`GxB_LIBRARY_COMPILE_TIME`	R	`char *`	time the library was compiled
`GxB_LIBRARY_URL`	R	`char *`	URL for the library
`GxB_API_DATE`	R	`char *`	C API release date
`GxB_API_ABOUT`	R	`char *`	about the C API
`GxB_API_URL`	R	`char *`	URL for the C API
`GxB_COMPILER_NAME`	R	`char *`	name of the compiler used to compile the library
`GxB_JIT_C_COMPILER_NAME`	R/W	`char *`	See Section 9
`GxB_JIT_C_COMPILER_FLAGS`	R/W	`char *`	”
`GxB_JIT_C_LINKER_FLAGS`	R/W	`char *`	”
`GxB_JIT_C_LIBRARIES`	R/W	`char *`	”
`GxB_JIT_C_CMAKE_LIBS`	R/W	`char *`	”
`GxB_JIT_C_PREFACE`	R/W	`char *`	”
`GxB_JIT_ERROR_LOG`	R/W	`char *`	”
`GxB_JIT_CACHE_PATH`	R/W	`char *`	”

int field	R/W	C type	description
GxB_BITMAP_SWITCH	R/W	void *	double array of size GxB_NBITMAP_SWITCH. See Section 10.10.3.
GxB_COMPILER_VERSION	R	void *	int32_t array of size 3. The version of the compiler used to compile the library.
GxB_PRINTF	W	void *	pointer to printf function for diagnostic output. See Section 10.2.1.
GxB_FLUSH	W	void *	pointer to flush function for diagnostic output. See Section 10.2.1.
GxB_MALLOC_FUNCTION	R	void *	malloc function
GxB_CALLOC_FUNCTION	R	void *	calloc function
GxB_REALLOC_FUNCTION	R	void *	realloc function
GxB_FREE_FUNCTION	R	void *	free function

10.2.1 Global diagnostic settings

`GrB_set (GrB_GLOBAL, ..., GxB_BURBLE)` controls the burble setting. It can also be controlled via `GrB.burble(b)` in the MATLAB/Octave interface.

```
GrB_set (GrB_GLOBAL, true,  GxB_BURBLE) ;  // enable burble
GrB_set (GrB_GLOBAL, false, GxB_BURBLE) ;  // disable burble
```

If enabled, SuiteSparse:GraphBLAS reports which internal kernels it uses, and how much time is spent. If you see the word `generic`, it means that SuiteSparse:GraphBLAS was unable to use its JIT kernels, or its faster kernels in `Source/FactoryKernels`, but used a generic kernel that relies on function pointers. This is done for user-defined types and operators when they cannot be used in the JIT, and when typecasting is performed. Generic kernels are typically slower than the JIT kernels or kernels in `Source/FactoryKernels`.

If you see a lot of `wait` statements, it may mean that a lot of time is spent finishing a matrix or vector. This may be the result of an inefficient use of the `setElement` and `assign` methods. If this occurs you might try changing the sparsity format of a vector or matrix to `GxB_BITMAP`, assuming there's enough space for it.

The following setting allows the user application to change the function used to print diagnostic output:

```
GrB_set (GrB_GLOBAL, (void *) printf, GxB_PRINTF, sizeof (void *)) ;
```

This also controls the output of the `GxB_*print` functions. By default this parameter is `NULL`, in which case the C11 `printf` function is used. The parameter is a function pointer with the same signature as the C11 `printf` function. The MATLAB/Octave interface to GraphBLAS sets it to `mexPrintf` so that GraphBLAS can print to the MATLAB/Octave Command Window.

After each call to the `printf` function, an optional `flush` function is called, which is `NULL` by default. If `NULL`, the function is not used. This can be changed with:

```
GrB_set (GrB_GLOBAL, (void *) flush, GxB_FLUSH, sizeof (void *)) ;
```

The `flush` function takes no arguments, and returns an `int` which is 0 if successful, or any nonzero value on failure (the same output as the C11 `fflush` function, except that `flush` has no inputs).

10.2.2 OpenMP parallelism

SuiteSparse:GraphBLAS is a parallel library, based on OpenMP. By default, all GraphBLAS operations will use up to the maximum number of threads specified by the `omp_get_max_threads` OpenMP function. For small problems, GraphBLAS may choose to use fewer threads, using two parameters: the maximum number of threads to use (which may differ from the `omp_get_max_threads` value), and a parameter called the `chunk`. Suppose `work` is a measure of the work an operation needs to perform (say the number of entries in the two input matrices for `GrB_eWiseAdd`). No more than `floor(work/chunk)` threads will be used (or one thread if the ratio is less than 1).

`GxB_NTHREADS` controls how many threads a method uses. By default (if set to zero, or `GrB_DEFAULT`), all available threads are used. The maximum available threads is controlled by the global setting, which is `omp_get_max_threads ( )` by default. If set to some positive integer `nthreads` less than this maximum, at most `nthreads` threads will be used.

`GxB_CHUNK` is a `double` value that controls how many threads a method uses for small problems. 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`.

```
int32_t nthreads_max = 40 ;
GrB_set (GrB_GLOBAL, nthreads_max, GxB_NTHREADS) ;
GrB_Scalar_new (&s, GrB_FP64) ;
GrB_Scalar_setElement (s, (double) 20000) ;
GrB_set (GrB_GLOBAL, s, GxB_CHUNK) ;
```

- Context: this object can be used to choose a different number of threads used in calls to GraphBLAS from different user threads, exploiting nested parallelism. Refer to Section 8. If a thread has engaged a context object, it ignores the global settings for `GxB_NTHREADS` and `GxB_CHUNK`, and uses the settings in its own context instead.

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 `GrB_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 `GrB_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 `GrB_set (GrB_GLOBAL, 4, GxB_NTHREADS)` is used, then the global setting is four threads, not eight.

GraphBLAS may be compiled without OpenMP, by setting `-DNOPENMP=1`. The library will be thread-safe, with one exception. `GrB_wait` is intended to provide thread-safety by flushing the cache of one user thread so the object can be safely read by another thread. This is accomplished with `pragma omp flush`, but if OpenMP is not available, this does nothing. If OpenMP is not available or `-DNOPEMP=1` is used, then user applications need to ensure their own thread safety when one user thread computes a result that is then read by another thread.

You can query GraphBLAS at run-time to ask if it was compiled with OpenMP:

```
bool have_openmp ;
GrB_get (GrB_GLOBAL, &have_openmp, GxB_LIBRARY_OPENMP) ;
if (!have_openmp) printf ("GraphBLAS not compiled with OpenMP\n") :
```

Compiling GraphBLAS without OpenMP is not recommended for installation in a package manager (Linux, conda-forge, spack, brew, vcpkg, etc).

10.2.3 Other global options

GrB_BLOCKING_MODE can only be queried by GrB_get; it cannot be modified by GrB_set. The mode is the value passed to GrB_init (blocking or non-blocking).

All threads in the same user application share the same global options, including hypersparsity, bitmap options, and CSR/CSC format determined by GrB_set, and the blocking mode determined by GrB_init. Specific format and hypersparsity parameters of each matrix are specific to that matrix and can be independently changed.

The GrB_LIBRARY_* and GxB_LIBRARY_* options can be used to query the current implementation of SuiteSparse:GraphBLAS. The GrB_API_* and GxB_API_* options can be used to query the current GraphBLAS C API Specification.

10.3 GrB_Type Options

```
GrB_Info GrB_get (GrB_Type t, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_Type t, char *     value, int f) ;
GrB_Info GrB_get (GrB_Type t, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_Type t, size_t *   value, int f) ;

GrB_Info GrB_set (GrB_Type t, char *     value, int f) ;
```

int field	R/W	C type	description
GrB_EL_TYPE_CODE	R	int32_t	type code (see GrB_Type_Code)
GrB_SIZE	R	size_t	sizeof the type
GrB_NAME	R/W1	char *	name of the type. For built-in types, this returns the GraphBLAS name ("GrB_FP32" for GrB_FP32, for example). For user-defined types, the name can be any string of any length. It is not used by the JIT. It can be set at most once.
GxB_JIT_C_NAME	R/W1	char *	This must be a valid name of a C type to enable its use in the JIT. For built-in types, this returns the C name of the type ("float" for GrB_FP32, for example). The length of the name can be at most GxB_MAX_NAME_LEN, including the nul terminating byte. It can be set at most once.
GxB_JIT_C_DEFINITION	R/W1	char *	type definition, as a C typedef; built-in types return an empty string. It can be set at most once.

Built-in types cannot be modified by GrB_set. User-defined types can be used without setting their name or definition, but they can be used in JIT kernels only when both the JIT C name and the definition are set.

To use the JIT, all operators, monoids, and semirings that access this type must be defined after the user-defined type has been given both a name and a definition. GraphBLAS can use an operator that uses a type without a name, but it cannot use the JIT, even if the type is given a name later on after the operator is created.

The size of the type can be returned as a size_t C scalar, or as a GrB_Scalar, normally of type GrB_UINT64, with the examples below.

```
size_t size ;
```

```
GrB_get (GrB_FP32, &size, GrB_SIZE) ;
assert (size == sizeof (float)) ;

GrB_Scalar s ;
GrB_Scalar_new (&s, GrB_UINT64) ;
GrB_get (GrB_FP32, s, GrB_SIZE) ;
GrB_Scalar_extractElement (&size, s) ;
assert (size == sizeof (float)) ;
```

10.4 GrB_UnaryOp Options

```
GrB_Info GrB_get (GrB_UnaryOp op, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_UnaryOp op, char *     value, int f) ;
GrB_Info GrB_get (GrB_UnaryOp op, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_UnaryOp op, size_t *   value, int f) ;

GrB_Info GrB_set (GrB_UnaryOp op, char *     value, int f) ;
```

int field	R/W	C type	description
GrB_INP0_TYPE_CODE	R	int32_t	input type code (see GrB_Type_code)
GrB_OUTP_TYPE_CODE	R	int32_t	output type code
GrB_INP0_TYPE_STRING	R	char *	name of the input type
GrB_OUTP_TYPE_STRING	R	char *	name of the output type
GrB_NAME	R/W1	char *	name of the operator. For built-in operators, this returns the GraphBLAS name ("GrB_LNOT" for GrB_LNOT, for example). For user-defined operators, the name can be any string of any length. It is not used by the JIT. It can be set at most once.
GxB_JIT_C_NAME	R/W1	char *	This must be a valid name of a C function to enable its use in the JIT. The length of the name can be at most GxB_MAX_NAME_LEN, including the nul terminating byte. It can be set at most once.
GxB_JIT_C_DEFINITION	R/W1	char *	definition for a user-defined operator, as a C function; built-in operators return an empty string. It can be set at most once.

Built-in operators cannot be modified by GrB_set. User-defined operators can be used without setting their name or definition, but they can be used in JIT kernels only when both the JIT C name and the definition are set.

10.5 GrB_IndexUnaryOp Options

```
GrB_Info GrB_get (GrB_IndexUnaryOp op, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_IndexUnaryOp op, char *     value, int f) ;
GrB_Info GrB_get (GrB_IndexUnaryOp op, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_IndexUnaryOp op, size_t *   value, int f) ;

GrB_Info GrB_set (GrB_IndexUnaryOp op, char *     value, int f) ;
```

int field	R/W	C type	description
GrB_INP0_TYPE_CODE	R	int32_t	1st input type code (see GrB_Type_code)
GrB_INP1_TYPE_CODE	R	int32_t	2nd input type code
GrB_OUTP_TYPE_CODE	R	int32_t	output type code
GrB_INP0_TYPE_STRING	R	char *	name of the 1st input type
GrB_INP1_TYPE_STRING	R	char *	name of the 2nd input type
GrB_OUTP_TYPE_STRING	R	char *	name of the output type
GrB_NAME	R/W1	char *	name of the operator. For built-in operators, this returns the GraphBLAS name ("GrB_TRIL" for GrB_TRIL, for example). For user-defined operators, the name can be any string of any length. It is not used by the JIT. It can be set at most once.
GxB_JIT_C_NAME	R/W1	char *	This must be a valid name of a C function to enable its use in the JIT. The length of the name can be at most GxB_MAX_NAME_LEN, including the nul terminating byte. It can be set at most once.
GxB_JIT_C_DEFINITION	R/W1	char *	definition for a user-defined operator, as a C function; built-in operators return an empty string. It can be set at most once.

Built-in operators cannot be modified by GrB_set. User-defined operators can be used without setting their name or definition, but they can be used in JIT kernels only when both the JIT C name and the definition are set.

10.6 GrB_BinaryOp Options

```
GrB_Info GrB_get (GrB_BinaryOp op, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_BinaryOp op, char *     value, int f) ;
GrB_Info GrB_get (GrB_BinaryOp op, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_BinaryOp op, size_t *   value, int f) ;

GrB_Info GrB_set (GrB_BinaryOp op, char *     value, int f) ;
```

int field	R/W	C type	description
GrB_INP0_TYPE_CODE	R	int32_t	1st input type code (see GrB_Type_code)
GrB_INP1_TYPE_CODE	R	int32_t	2nd input type code
GrB_OUTP_TYPE_CODE	R	int32_t	output type code
GxB_THETA_TYPE_CODE	R	int32_t	Θ type code, if any
GrB_INP0_TYPE_STRING	R	char *	name of the 1st input type
GrB_INP1_TYPE_STRING	R	char *	name of the 2nd input type
GrB_OUTP_TYPE_STRING	R	char *	name of the output type
GxB_THETA_TYPE_STRING	R	char *	name of the Θ type, if any
GrB_NAME	R/W1	char *	name of the operator. For built-in operators, this returns the GraphBLAS name ("GrB_LOR" for GrB_LOR, for example). For user-defined operators, the name can be any string of any length. It is not used by the JIT. It can be set at most once.
GxB_JIT_C_NAME	R/W1	char *	This must be a valid name of a C function to enable its use in the JIT. The length of the name can be at most GxB_MAX_NAME_LEN, including the nul terminating byte. It can be set at most once.
GxB_JIT_C_DEFINITION	R/W1	char *	definition for a user-defined operator, as a C function; built-in operators return an empty string. It can be set at most once.
GxB_THETA	R	GrB_Scalar	value of Theta, if any. The type of the GrB_Scalar must match the Theta type of the underlying index-binary operator exactly.

Built-in operators cannot be modified by GrB_set. User-defined operators can be used without setting their name or definition, but they can be used in JIT kernels only when both the JIT C name and the definition are set.

To use the JIT, all monoids and semirings that access this binary operator must be defined after the user-defined operator has been given both a name and a definition. GraphBLAS can use a monoid or semiring that uses a binary operator without a name, but it cannot use the JIT, even if the operator is given a name later on after the operator is created.

The `*THETA*` options can only be used in the binary operator was created by `GxB_BinaryOp_new_IndexOp`.

10.7 GxB_IndexBinaryOp Options

```
GrB_Info GrB_get (GxB_IndexBinaryOp op, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GxB_IndexBinaryOp op, char *     value, int f) ;
GrB_Info GrB_get (GxB_IndexBinaryOp op, int32_t *  value, int f) ;
GrB_Info GrB_get (GxB_IndexBinaryOp op, size_t *   value, int f) ;

GrB_Info GrB_set (GxB_IndexBinaryOp op, char *     value, int f) ;
```

int field	R/W	C type	description
GrB_INP0_TYPE_CODE	R	int32_t	1st input type code (see GrB_Type_code)
GrB_INP1_TYPE_CODE	R	int32_t	2nd input type code
GrB_OUTP_TYPE_CODE	R	int32_t	output type code
GxB_THETA_TYPE_CODE	R	int32_t	Θ type code
GrB_INP0_TYPE_STRING	R	char *	name of the 1st input type
GrB_INP1_TYPE_STRING	R	char *	name of the 2nd input type
GrB_OUTP_TYPE_STRING	R	char *	name of the output type
GxB_THETA_TYPE_STRING	R	char *	name of the Θ type
GrB_NAME	R/W1	char *	name of the operator. For user-defined operators, the name can be any string of any length. It is not used by the JIT. It can be set at most once.
GxB_JIT_C_NAME	R/W1	char *	This must be a valid name of a C function to enable its use in the JIT. The length of the name can be at most GxB_MAX_NAME_LEN, including the nul terminating byte. It can be set at most once.
GxB_JIT_C_DEFINITION	R/W1	char *	definition for a user-defined operator, as a C function; built-in operators return an empty string. It can be set at most once.

There are no built-in index-binary operators, but if there are in the future, they will not be be modified by GrB_set. User-defined operators can be used without setting their name or definition, but they can be used in JIT kernels only when both the JIT C name and the definition are set.

To use the JIT, all semirings that access this index-binary operator must be defined after the user-defined operator has been given both a name and a definition. GraphBLAS can use a semiring that uses a binary operator without a name, but it cannot use the JIT, even if the operator is given a name later on after the operator is created.

10.8 GrB_Monoid Options

```
GrB_Info GrB_get (GrB_Monoid monoid, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_Monoid monoid, char *     value, int f) ;
GrB_Info GrB_get (GrB_Monoid monoid, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_Monoid monoid, size_t *   value, int f) ;
GrB_Info GrB_get (GrB_Monoid monoid, void *     value, int f) ;

GrB_Info GrB_set (GrB_Monoid monoid, char *     value, int f) ;
```

int field	R/W	C type	description
GrB_INP0_TYPE_CODE	R	int32_t	1st input type code (see GrB_Type_code)
GrB_INP1_TYPE_CODE	R	int32_t	2nd input type code
GrB_OUTP_TYPE_CODE	R	int32_t	output type code
GrB_INP0_TYPE_STRING	R	char *	name of the 1st input type
GrB_INP1_TYPE_STRING	R	char *	name of the 2nd input type
GrB_OUTP_TYPE_STRING	R	char *	name of the output type
GrB_NAME	R/W1	char *	name of the monoid. For built-in monoids, this returns the GraphBLAS name ("GrB_LOR_MONOID_BOOL" for GrB_LOR_MONOID_BOOL, for example). For user-defined monoids, the name can be any string of any length. It is not used by the JIT. It can be set at most once.
GxB_MONOID_IDENTITY	R	GrB_Scalar	identity value of the monoid. The type of the GrB_Scalar must match the monoid type exactly.
GxB_MONOID_TERMINAL	R	GrB_Scalar	terminal value of a terminal monoid. The type of the GrB_Scalar must match the monoid type exactly. If the monoid is not terminal, the GrB_Scalar is returned with no entry.
GxB_MONOID_OPERATOR	R	void *	binary operator of the monoid, as a GrB_BinaryOp

Built-in monoids cannot be modified by GrB_set.

For GxB_MONOID_OPERATOR, the op is returned as an alias, not as a new object. For example, if a monoid is created with a user-defined binary operator, the following usage returns a shallow copy of the operator:

```
GrB_BinaryOp binop ;
GrB_BinaryOp_new (&binop, func, GrB_BOOL, GrB_BOOL, GrB_BOOL) ;
GrB_Monoid monoid ;
GrB_Monoid_new (&monoid, binop, (bool) false) ;
```

With the above objects defined, the following two code snippets do the same thing:

```
// getting an alias to the binary operator directly:
GrB_BinaryOp op ;
op = binop ;

// getting an alias to the binary operator using GrB_get:
GrB_BinaryOp op ;
GrB_get (monoid, (void *) &op, GxB_MONOID_OPERATOR) ;
assert (op == binop) ;
```

As a result, it is not valid to free both the `op` and the `binop`, since they are the same object. This usage returns the built-in `GrB_LOR` operator of the corresponding built-in monoid:

```
GrB_BinaryOp op ;
GrB_get (GrB_LOR_MONOID, (void *) &op, GxB_MONOID_OPERATOR) ;
assert (op == GrB_LOR) ;
```

10.9 GrB_Semiring Options

```
GrB_Info GrB_get (GrB_Semiring semiring, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_Semiring semiring, char *     value, int f) ;
GrB_Info GrB_get (GrB_Semiring semiring, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_Semiring semiring, size_t *   value, int f) ;
GrB_Info GrB_get (GrB_Semiring semiring, void *     value, int f) ;

GrB_Info GrB_set (GrB_Semiring semiring, GrB_Scalar value, int f) ;
```

`int field`	R/W	C type	description
`GrB_INP0_TYPE_CODE`	R	`int32_t`	1st input type code (see `GrB_Type_code`) of the multiplicative operator
`GrB_INP1_TYPE_CODE`	R	`int32_t`	2nd input type code of the multiplicative operator
`GrB_OUTP_TYPE_CODE`	R	`int32_t`	output type code of the multiplicative operator, and the monoid type.
`GxB_THETA_TYPE_CODE`	R	`int32_t`	Θ type code, if any
`GrB_INP0_TYPE_STRING`	R	`char *`	name of the 1st input type of the multiplicative operator
`GrB_INP1_TYPE_STRING`	R	`char *`	name of the 2nd input type of the multiplicative operator
`GrB_OUTP_TYPE_STRING`	R	`char *`	name of the output type of the multiplicative operator, and the monoid type.
`GxB_THETA_TYPE_STRING`	R	`char *`	name of the Θ type, if any
`GrB_NAME`	R/W1	`char *`	name of the semiring. For built-in semirings, this returns the GraphBLAS name (`"GrB_LOR_LAND_SEMIRING_BOOL"` for `GrB_LOR_LAND_SEMIRING_BOOL`, for example). For user-defined semirings, the name can be any string of any length. It is not used by the JIT. It can be set at most once.
`GxB_THETA`	R	`GrB_Scalar`	value of `Theta`, if any. The type of the `GrB_Scalar` must match the `Theta` type of the underlying index-binary operator exactly.

int field	R/W	C type	description
GxB_MONOID_IDENTITY	R	GrB_Scalar	identity value of the monoid. The type of the GrB_Scalar must match the monoid type exactly.
GxB_MONOID_TERMINAL	R	GrB_Scalar	terminal value of a terminal monoid. The type of the GrB_Scalar must match the monoid type exactly. If the monoid is not terminal, the GrB_Scalar is returned with no entry.
GxB_MONOID_OPERATOR	R	void *	binary operator of the monoid, as a GrB_BinaryOp; See Section 10.8
GxB_SEMIRING_MONOID	R	void *	monoid of the semiring, as a GrB_Monoid
GxB_SEMIRING_MULTIPLY	R	void *	multiplicative operator of the semiring, as a GrB_BinaryOp

Built-in semirings cannot be modified by GrB_set.

The GxB_SEMIRING_MONOID option returns the GrB_Monoid of the semiring. The GxB_SEMIRING_MULTIPLY option returns the GrB_BinaryOp for the multiplicative operator of the semiring. For example:

```
// getting an alias to the monoid and multiply operator using GrB_get:
GrB_BinaryOp op ;
GrB_Monoid mon ;
GrB_Semiring semiring = GrB_PLUS_TIMES_FP32 ;
GrB_get (semiring, (void *) &mon, GxB_SEMIRING_MONOID) ;
GrB_get (semiring, (void *) &op, GxB_SEMIRING_MULTIPLY) ;
assert (op == GrB_TIMES_FP32) ;
assert (mon == GrB_PLUS_MONOID_FP32) ;
```

The binary op and monoid returned are aliases, not new objects.

The *THETA* options can only be used in the multiplicative binary operator of the semiring was created by GxB_BinaryOp_new_IndexOp.

10.10 GrB_Matrix Options

```
GrB_Info GrB_get (GrB_Matrix A, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_Matrix A, char *     value, int f) ;
GrB_Info GrB_get (GrB_Matrix A, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_Matrix A, size_t *   value, int f) ;

GrB_Info GrB_set (GrB_Matrix A, GrB_Scalar value, int f) ;
GrB_Info GrB_set (GrB_Matrix A, char *     value, int f) ;
GrB_Info GrB_set (GrB_Matrix A, int32_t    value, int f) ;
```

`int field`	R/W	C type	description
`GrB_STORAGE_ORIENTATION_HINT`	R/W	`int32_t`	See `GrB_Orientation`, and Section 10.10.1.
`GrB_EL_TYPE_CODE`	R	`int32_t`	matrix type
`GxB_SPARSITY_CONTROL`	R/W	`int32_t`	See Section 10.10.4
`GxB_SPARSITY_STATUS`	R	`int32_t`	See Section 10.10.4
`GrB_NAME`	R/W	`char *`	name of the matrix. This can be set any number of times.
`GrB_EL_TYPE_STRING`	R	`char *`	name of the type of the matrix.
`GxB_JIT_C_NAME`	R	`char *`	JIT C name of the type of the matrix.
`GxB_HYPER_SWITCH`	R/W	`double`	See Section 10.10.2
`GxB_BITMAP_SWITCH`	R/W	`double`	See Section 10.10.3

10.10.1 Storing a matrix by row or by column

The GraphBLAS `GrB_Matrix` is entirely opaque to the user application, and the GraphBLAS API does not specify how the matrix should be stored. However, choices made in how the matrix is represented in a particular implementation, such as SuiteSparse:GraphBLAS, can have a large impact on performance.

Many graph algorithms are just as fast in any format, but some algorithms are much faster in one format or the other. For example, suppose the user application stores a directed graph as a matrix `A`, with the edge (i, j) represented as the value `A(i,j)`, and the application makes many accesses to the ith row of the matrix, with `GrB_Col_extract (w,...,A,GrB_ALL,...,i,desc)` with the transposed descriptor (`GrB_INP0` set to `GrB_TRAN`). If the matrix is stored by column this can be extremely slow, just like the expression `w=A(i,:)` in MATLAB, where `i` is a scalar. Since this is a typical use-case in graph algorithms, the default format in SuiteSparse:GraphBLAS is to store its matrices by row, in Compressed Sparse Row format (CSR).

MATLAB stores its sparse matrices by column, in "non-hypersparse" format, in what is called the Compressed Sparse Column format, or CSC for short. An `m`-by-`n` matrix in MATLAB is represented as a set of `n` column vectors, each with a sorted list of row indices and values of the nonzero entries in that column. As a result, `w=A(:,j)` is very fast in MATLAB, since the result is already held in the data structure a single list, the jth column vector. However, `w=A(i,:)` is very slow in MATLAB, since every column in the matrix has to be searched to see if it contains row `i`. In MATLAB, if many such accesses are made, it is much better to transpose the matrix (say `AT=A'`) and then use `w=AT(:,i)` instead. This can have a dramatic impact on the performance of MATLAB.

Likewise, if `u` is a very sparse column vector and `A` is stored by column, then `w=u'*A` (via `GrB_vxm`) is slower than `w=A*u` (via `GrB_mxv`). The opposite is true if the matrix is stored by row.

SuiteSparse:GraphBLAS stores its matrices by row, by default (with one exception described below). However, it can also be instructed to store any selected matrices, or all matrices, by column instead (just like MATLAB), so that `w=A(:,j)` (via `GrB_Col_extract`) is very fast. The change in data format has no effect on the result, just the time and memory usage. To use a column-oriented format by default, the following can be done in a user application that tends to access its matrices by column.

```
GrB_init (...) ;
// just after GrB_init: do the following:
GrB_set (GrB_GLOBAL, GrB_COLMAJOR, GrB_STORAGE_ORIENTATION_HINT) ;
```

If this is done, and no other `GrB_set` calls are made with `GrB_STORAGE_ORIENATION_HINT`, all matrices will be stored by column. The default format is `GrB_ROWMAJOR`.

All vectors (`GrB_Vector`) are held by column, and this cannot be changed.

By default, matrices of size `m-by-1` are held by column, regardless of the global setting described above. Matrices of size `1-by-n` with `n` not equal to 1 are held by row, regardless of the global setting. The global setting only affects matrices with both `m > 1` and `n > 1`. Empty matrices (`0-by-0`) are also controlled by the global setting.

After creating a matrix with `GrB_Matrix_new (&A, ...)`, its format can be changed arbitrarily with:

```
GrB_set (A, GrB_COLMAJOR, GrB_STORAGE_ORIENTATION_HINT) ;
GrB_set (A, GrB_ROWMAJOR, GrB_STORAGE_ORIENTATION_HINT) ;
```

If set to other values (GrB_BOTH or GrB_UNKNOWN), the format is changed to GrB_ROWMAJOR.

With this setting, even an m-by-1 matrix can then be changed to be held by row, for example. Likewise, once a 1-by-n matrix is created, it can be converted to column-oriented format.

10.10.2 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 $k \le n$ 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 `GrB_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 ;
GrB_set (A, hyper_switch, GxB_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 `GrB_ROWMAJOR` or `GrB_COLMAJOR` format, as determined by the last call to `GrB_set(GrB_GLOBAL, ..., GrB_STORAGE_ORIENTATION_HINT,...)` 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`.

10.10.3 Bitmap matrices

By default, SuiteSparse:GraphBLAS switches between all four formats (hypersparse, sparse, bitmap, and full) automatically. Let $d = |\mathbf{A}|/mn$ for an m-by-n matrix $\mathbf{A}$ with $|\mathbf{A}|$ entries. If the matrix is currently in sparse or hypersparse format, and is modified so that d exceeds a given threshold, it is converted into bitmap format. The default threshold is controlled by the `GxB_BITMAP_SWITCH` setting, which can be set globally, or for a particular matrix or vector.

The default value of the switch to bitmap format depends on $\min(m, n)$, for a matrix of size m-by-n. For the global setting, the bitmap switch is a `double` array of size `GxB_NBITMAP_SWITCH`. The defaults are given below:

parameter	default	matrix sizes
`bitmap_switch [0]`	0.04	$\min(m, n) = 1$ (and all vectors)
`bitmap_switch [1]`	0.05	$\min(m, n) = 2$
`bitmap_switch [2]`	0.06	$\min(m, n) = 3$ to 4
`bitmap_switch [3]`	0.08	$\min(m, n) = 5$ to 8
`bitmap_switch [4]`	0.10	$\min(m, n) = 9$ to 16
`bitmap_switch [5]`	0.20	$\min(m, n) = 17$ to 32
`bitmap_switch [6]`	0.30	$\min(m, n) = 33$ to 64
`bitmap_switch [7]`	0.40	$\min(m, n) > 64$

That is, by default a `GrB_Vector` is held in bitmap format if its density exceeds 4%. To change the global settings, do the following:

```
double bswitch [GxB_NBITMAP_SWITCH] = { 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 } ;
GrB_set (GrB_GLOBAL, (void *) bswitch, GxB_BITMAP_SWITCH,
    GxB_NBITMAP_SWITCH * sizeof (double)) ;
```

If the matrix is currently in bitmap format, it is converted to full if all entries are present, or to sparse/hypersparse if d drops below $b/2$, if its bitmap switch is b. A matrix or vector with d between $b/2$ and b remains in its current format.

10.10.4 Sparsity status

The sparsity status of a matrix can be queried with the following, which returns a value of `GxB_HYPERSPARSE` (1), `GxB_SPARSE` (2), `GxB_BITMAP` (4), or `GxB_FULL` (8).

```
int32_t sparsity ;
GrB_get (A, &sparsity, GxB_SPARSITY_STATUS) ;
```

The sparsity format of a matrix can be controlled with the field set to GxB_SPARSITY_CONTROL, for which the value can be any mix (a sum or bitwise or) of GxB_HYPERSPARSE, GxB_SPARSE, GxB_BITMAP, and GxB_FULL. By default, a matrix or vector can be held in any format, with the default setting GxB_AUTO_SPARSITY, which is equal to GxB_HYPERSPARSE + GxB_SPARSE + GxB_BITMAP + GxB_FULL (15). To enable a matrix to take on just GxB_SPARSE or GxB_FULL formats, but not GxB_HYPERSPARSE or GxB_BITMAP, for example, use the following:

```
GrB_set (A, GxB_SPARSE + GxB_FULL, GxB_SPARSITY_CONTROL) ;
```

In this case, SuiteSparse:GraphBLAS will hold the matrix in sparse format (CSR or CSC, depending on its GrB_STORAGE_ORIENTATION_HINT), unless all entries are present, in which case it will be converted to full format.

Only the least significant 4 bits of the sparsity control are considered, so the formats can be bitwise negated. For example, to allow for any format except full:

```
GrB_set (A, ~GxB_FULL, GxB_SPARSITY_CONTROL) ;
```

10.11 GrB_Vector Options

```
GrB_Info GrB_get (GrB_Vector v, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_Vector v, char *     value, int f) ;
GrB_Info GrB_get (GrB_Vector v, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_Vector v, size_t *   value, int f) ;

GrB_Info GrB_set (GrB_Vector v, GrB_Scalar value, int f) ;
GrB_Info GrB_set (GrB_Vector v, char *     value, int f) ;
GrB_Info GrB_set (GrB_Vector v, int32_t    value, int f) ;
```

`int field`	R/W	C type	description
`GrB_EL_TYPE_CODE`	R	`int32_t`	vector type
`GxB_SPARSITY_CONTROL`	R/W	`int32_t`	See Section 10.10.4
`GxB_SPARSITY_STATUS`	R	`int32_t`	See Section 10.10.4
`GrB_NAME`	R/W	`char *`	name of the vector.
`GrB_EL_TYPE_STRING`	R	`char *`	name of the type of the vector.
`GxB_JIT_C_NAME`	R	`char *`	JIT C name of the type of the vector.
`GxB_BITMAP_SWITCH`	R/W	`double`	See Section 10.10.3

See Section 10.10; a `GrB_Vector` is treated as if it were an n-by-1 matrix, and is always in column major form. It is never hypersparse.

10.12 GrB_Scalar Options

```
GrB_Info GrB_get (GrB_Scalar s, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_Scalar s, char *     value, int f) ;
GrB_Info GrB_get (GrB_Scalar s, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_Scalar s, size_t *   value, int f) ;

GrB_Info GrB_set (GrB_Scalar s, GrB_Scalar value, int f) ;
GrB_Info GrB_set (GrB_Scalar s, char *     value, int f) ;
GrB_Info GrB_set (GrB_Scalar s, int32_t    value, int f) ;
```

`int field`	R/W	C type	description
`GrB_EL_TYPE_CODE`	R	`int32_t`	scalar type
`GxB_SPARSITY_STATUS`	R	`int32_t`	See Section 10.10.4
`GrB_NAME`	R/W	`char *`	name of the scalar.
`GrB_EL_TYPE_STRING`	R	`char *`	name of the type of the scalar.
`GxB_JIT_C_NAME`	R	`char *`	JIT C name of the type of the scalar.

See Section 10.10; a `GrB_Scalar` is treated as if it were a 1-by-1 matrix, and is always in column major form. It is never hypersparse.

10.13 GrB_Descriptor Options

```
GrB_Info GrB_get (GrB_Descriptor desc, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GrB_Descriptor desc, char *     value, int f) ;
GrB_Info GrB_get (GrB_Descriptor desc, int32_t *  value, int f) ;
GrB_Info GrB_get (GrB_Descriptor desc, size_t *   value, int f) ;

GrB_Info GrB_set (GrB_Descriptor desc, GrB_Scalar value, int f) ;
GrB_Info GrB_set (GrB_Descriptor desc, char *     value, int f) ;
GrB_Info GrB_set (GrB_Descriptor desc, int32_t    value, int f) ;
```

`int field`	R/W	C type	description
`GrB_OUTP`	R/W	`int32_t`	`GrB_DEFAULT` or `GrB_REPLACE`
`GrB_MASK`	R/W	`int32_t`	`GrB_DEFAULT`, `GrB_COMP`, `GrB_STRUCTURE`, or `GrB_COMP_STRUCTURE`
`GrB_INP0`	R/W	`int32_t`	`GrB_DEFAULT` or `GrB_TRAN`
`GrB_INP1`	R/W	`int32_t`	`GrB_DEFAULT` or `GrB_TRAN`
`GxB_AxB_METHOD`	R/W	`int32_t`	Method used by `GrB_mxm` (`GrB_DEFAULT`, `GxB_AxB_GUSTAVSON`. `GxB_AxB_HASH`, `GxB_AxB_SAXPY`, or `GxB_AxB_DOT`).
`GxB_SORT`	R/W	`int32_t`	if true, `GrB_mxm` returns its output in sorted form.
`GxB_COMPRESSION`	R/W	`int32_t`	compression method for serialize methods.
`GxB_IMPORT`	R/W	`int32_t`	`GxB_FAST_IMPORT` or `GxB_SECURE_IMPORT` for `GxB*_pack*` methods.
`GrB_NAME`	R/W	`char *`	name of the descriptor. This can be set any number of times for user-defined descriptors. Built-in descriptors have the same name as the variable name (`"GrB_DESC_T1"` for the `GrB_DESC_T1` descriptor, for example)

The following table describes each option. See Section 6.14 for more details.

Descriptor field	Default	Non-default
GrB_OUTP	GrB_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	GrB_DEFAULT: The Mask is not complemented. Mask(i,j)=1 means the value C_{ij} can be modified by the operation, while Mask(i,j)=0 means the value C_{ij} shall not be modified by the operation.	GrB_COMP: The Mask is complemented. Mask(i,j)=0 means the value C_{ij} can be modified by the operation, while Mask(i,j)=1 means the value C_{ij} shall not be modified by the operation. GrB_STRUCTURE: The values of the Mask are ignored. If Mask(i,j) is an entry in the Mask matrix, it is treated as if Mask(i,j)=1. The two options GrB_COMP and GrB_STRUCTURE can be combined, with two subsequent calls, or with a single call with the setting GrB_COMP+GrB_STRUCTURE.
GrB_INP0	GrB_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	GrB_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.
GxB_AxB_METHOD	GrB_DEFAULT: The method for C=A*B is selected automatically.	GxB_AxB_*method*: The selected method is used to compute C=A*B.
GxB_SORT	GrB_DEFAULT: The computation of C=A*B may leave C in a jumbled state; GrB_wait will finalize the matrix.	any nonzero value: C=A*B always returns C in final, sorted form.
GxB_COMPRESION	GrB_DEFAULT: Serialize methods will use the default method, ZSTD (level 1)	See Section 6.11
GxB_IMPORT	GrB_DEFAULT: fast import	GxB_SECURE_IMPORT: secure import

10.14 GxB_Context Options

```
GrB_Info GrB_get (GxB_Context Context, GrB_Scalar value, int f) ;
GrB_Info GrB_get (GxB_Context Context, char *     value, int f) ;
GrB_Info GrB_get (GxB_Context Context, int32_t *  value, int f) ;
GrB_Info GrB_get (GxB_Context Context, size_t *   value, int f) ;

GrB_Info GrB_set (GxB_Context Context, GrB_Scalar value, int f) ;
GrB_Info GrB_set (GxB_Context Context, char *     value, int f) ;
GrB_Info GrB_set (GxB_Context Context, int32_t    value, int f) ;
```

`int field`	R/W	C type	description
`GxB_NTHREADS`	R/W	`int32_t`	number of OpenMP threads to use; See Section 10.2.2
`GxB_CHUNK`	R/W	`double`	chunk factor for task creation; See Section 10.2.2
`GrB_NAME`	R/W	`char *`	name of the context. This can be set any number of times for user-defined contexts. Built-in contexts have the same name as the variable name (`"GxB_CONTEXT_WORLD"` for the `GxB_CONTEXT_WORLD` context, for example)

NOTE: the non-polymorphic names for this method are `GxB_Context_get_[KIND]` and `GxB_Context_set_[KIND]`, where `KIND` can be: `Scalar` (for `GrB_Scalar`), `String` (for `char *`), `INT` (for `int32_t`), and `SIZE` (for `size_t *` for `GrB_get` only), and `VOID` (for `void *`). The non-polymorphic suffix of `INT` is used here instead of `INT32` because `GxB_Context_*_INT32` appear as historical methods in version v8.0 and earlier, which are now deprecated.

For the `int32_t` type, the use of the polymorphic `GrB_set` and `GrB_get` accesses the correct version of this method. When using non-polymorphic methods, the use of `GxB_Context_get_INT` and `GxB_Context_set_INT` is recommended.

10.15 Options for inspecting a serialized blob

```
GrB_Info GrB_get (const void *blob, GrB_Scalar value, int f, size_t blobsize) ;
GrB_Info GrB_get (const void *blob, char *     value, int f, size_t blobsize) ;
GrB_Info GrB_get (const void *blob, int32_t *  value, int f, size_t blobsize) ;
GrB_Info GrB_get (const void *blob, size_t *   value, int f, size_t blobsize) ;
```

int field	R/W	C type	description
GrB_STORAGE_ORIENTATION_HINT	R	int32_t	See GrB_Orientation, and Section 10.10.1.
GrB_EL_TYPE_CODE	R	int32_t	type of matrix in the blob
GxB_SPARSITY_CONTROL	R	int32_t	See Section 10.10.4
GxB_SPARSITY_STATUS	R	int32_t	See Section 10.10.4
GrB_NAME	R	char *	name of the matrix in the blob.
GrB_EL_TYPE_STRING	R	char *	name of the type of the matrix in the blob.
GxB_JIT_C_NAME	R	char *	JIT C name of the type of the matrix in the blob.
GxB_HYPER_SWITCH	R	double	See Section 10.10.2
GxB_BITMAP_SWITCH	R	double	See Section 10.10.3

The GrB_Matrix_serialize and GxB_Matrix_serialize methods create a *blob* as a single array of bytes that contains all content of a GrB_Matrix. These GrB_get methods can query a blob for the same values that can be queried for a GrB_Matrix. The blob cannot be modified by GrB_set.

Note that these GrB_get methods add a fourth parameter, the size of the blob. All other GrB_get methods have just three parameters: the object, the value, and the field.

11 SuiteSparse:GraphBLAS Colon and Index Notation

MATLAB/Octave 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/Octave statements:

```
I = [2 1 4] ;
J = [3 5] ;
C = A (I,J) ;
```

which creates the 3-by-2 matrix `C` as follows:

$$C = \left[\begin{array}{cc} a_{2,3} & a_{2,5} \\ a_{1,3} & a_{1,5} \\ a_{4,3} & a_{4,5} \end{array} \right]$$

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 12) 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 12.10 for details.

2. To specify all rows of a matrix, use `I = GrB_ALL`. The parameter `ni` is ignored. This is equivalent to `C=A(:,J)` in MATLAB. In GraphBLAS, this is the sequence `0:(m-1)` if `A` has `m` rows, with length `|I|=m`. If `J` is used the columns of an `m`-by-`n` matrix, then `J=GrB_ALL` refers to all columns, and is the sequence `0:(n-1)`, of length `|J|=n`.

SPEC: If `I` or `J` are `GrB_ALL`, the specification requires that `ni` be passed in as `m` (the number of rows) and `nj` be passed in as `n`. Any other value is an error. SuiteSparse:GraphBLAS ignores these scalar inputs and treats them as if they are equal to their only possible correct value.

3. To specify a contiguous range of indices, such as `I=10:20` in MATLAB, the array `I` has size 2, and `ni` is passed to SuiteSparse:GraphBLAS as the special value `ni = GxB_RANGE`. The beginning index is `I[GxB_BEGIN]` and the ending index is `I[GxB_END]`. Both values must be non-negative since `GrB_Index` is an unsigned integer (`uint64_t`). The value of `I[GxB_INC]` is ignored.

```
// to specify I = 10:20
GrB_Index I [2], ni = GxB_RANGE ;
I [GxB_BEGIN] = 10 ;      // the start of the sequence
I [GxB_END  ] = 20 ;      // the end of the sequence
```

   Let $b =$ `I[GxB_BEGIN]`, let $e =$ `I[GxB_END]`, The sequence has length zero if $b > e$; otherwise the length is $|I| = (e - b) + 1$.

4. To specify a strided range of indices with a non-negative stride, such as `I=3:2:10`, the array `I` has size 3, and `ni` has the special value `GxB_STRIDE`. This is the sequence 3, 5, 7, 9, of length 4. Note that 10 does not appear in the list. The end point need not appear if the increment goes past it.

```
// to specify I = 3:2:10
GrB_Index I [3], ni = GxB_STRIDE ;
I [GxB_BEGIN ] = 3 ;      // the start of the sequence
I [GxB_INC   ] = 2 ;      // the increment
I [GxB_END   ] = 10 ;     // the end of the sequence
```

   The `GxB_STRIDE` sequence is the same as the `List` generated by the following for loop:

```
int64_t k = 0 ;
GrB_Index *List = (a pointer to an array of large enough size)
for (int64_t i = I [GxB_BEGIN] ; i <= I [GxB_END] ; i += I [GxB_INC])
{
    // i is the kth entry in the sequence
    List [k++] = i ;
}
```

Then passing the explicit array List and its length ni=k has the same effect as passing in the array I of size 3, with ni=GxB_STRIDE. The latter is simply much faster to produce, and much more efficient for SuiteSparse:GraphBLAS to process.

Let $b = $ I[GxB_BEGIN], let $e = $ I[GxB_END], and let $\Delta = $ I[GxB_INC]. The sequence has length zero if $b > e$ or $\Delta = 0$. Otherwise, the length of the sequence is

$$|I| = \left\lfloor \frac{e-b}{\Delta} \right\rfloor + 1$$

5. In MATLAB notation, if the stride is negative, the sequence is decreasing. For example, 10:-2:1 is the sequence 10, 8, 6, 4, 2, in that order. In SuiteSparse:GraphBLAS, use ni = GxB_BACKWARDS, with an array I of size 3. The following example specifies defines the equivalent of the MATLAB expression 10:-2:1 in SuiteSparse:GraphBLAS:

```
// to specify I = 10:-2:1
GrB_Index I [3], ni = GxB_BACKWARDS ;
I [GxB_BEGIN ] = 10 ;     // the start of the sequence
I [GxB_INC   ] = 2 ;      // the magnitude of the increment
I [GxB_END   ] = 1 ;      // the end of the sequence
```

The value -2 cannot be assigned to the GrB_Index array I, since that is an unsigned type. The signed increment is represented instead with the special value ni = GxB_BACKWARDS. The GxB_BACKWARDS sequence is the same as generated by the following for loop:

```
int64_t k = 0 ;
GrB_Index *List = (a pointer to an array of large enough size)
for (int64_t i = I [GxB_BEGIN] ; i >= I [GxB_END] ; i -= I [GxB_INC])
{
    // i is the kth entry in the sequence
    List [k++] = i ;
}
```

Let $b = $ I[GxB_BEGIN], let $e = $ I[GxB_END], and let $\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.

12 GraphBLAS Operations

The next sections define each of the GraphBLAS operations, also listed in the table below.

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

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`.

12.1 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 GrB_COLMAJOR format, and B is extremely sparse but A is not as sparse. Then computing C=A*B is very fast, and much faster than when A is extremely sparse. For example, if A is square and B is a column vector that is all nonzero except for one entry B(j,0)=1, then C=A*B is the same as extracting column A(:,j). This is very fast if A is stored by column but slow if A is stored by row. If A is a sparse row with a single entry A(0,i)=1, then C=A*B is the same as extracting row B(i,:). This is fast if B is stored by row but slow if B is stored by column.

If the user application needs to repeatedly extract rows and columns from a matrix, whether by matrix multiplication or by GrB_extract, then keep two copies: one stored by row, and other by column, and use the copy that results in the fastest computation.

By default, GrB_mxm, GrB_mxv, GrB_vxm, and GrB_reduce (to vector) can return their result in a jumbled state, with the sort left pending. It can sometimes be faster for these methods to do the sort as they compute their result. Use the GxB_SORT descriptor setting to select this option. Refer to Section 6.14 for details.

12.2 GrB_vxm: vector-matrix multiply

```
GrB_Info GrB_vxm                    // w'<mask> = accum (w, u'*A)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_Semiring semiring,    // defines '+' and '*' for u'*A
    const GrB_Vector u,             // first input:  vector u
    const GrB_Matrix A,             // second input: matrix A
    const GrB_Descriptor desc       // descriptor for w, mask, and A
) ;
```

`GrB_vxm` multiplies a row vector `u'` times a matrix `A`. The matrix `A` may be first transposed according to `desc` (as the second input, `GrB_INP1`); the column vector `u` is never transposed via the descriptor. The inputs `u` and `A` are typecasted to match the `xtype` and `ytype` inputs, respectively, of the multiply operator of the `semiring`. Next, an intermediate column vector `t=A'*u` is computed on the `semiring` using the same method as `GrB_mxm`. Finally, the column vector `t` is typecasted from the `ztype` of the multiply operator of the `semiring` into the type of `w`, and the results are written back into `w` using the optional accumulator `accum` and `mask`.

The last step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

Performance considerations: If the `GrB_STORAGE_ORIENTATION_HINT` of `A` is `GrB_ROWMAJOR`, 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 `GrB_STORAGE_ORIENTATION_HINT` of `A` is `GrB_COLMAJOR`, 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.

12.3 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 `GrB_COLMAJOR`. 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 `GrB_COLMAJOR` format. If an application needs to perform `A'*u` repeatedly where `u` is very sparse, then use the `GrB_ROWMAJOR` format for `A` instead.

12.4 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.

12.4.1 GrB_Vector_eWiseMult: element-wise vector multiply

```
GrB_Info GrB_eWiseMult              // w<mask> = accum (w, u.*v)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const <operator> multiply,      // defines '.*' for t=u.*v
    const GrB_Vector u,             // first input:  vector u
    const GrB_Vector v,             // second input: vector v
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_eWiseMult` computes the element-wise "multiplication" of two vectors `u` and `v`, element-wise using any binary operator (not just times). The vectors are not transposed via the descriptor. The vectors `u` and `v` are first typecasted into the first and second inputs of the `multiply` operator. Next, a column vector `t` is computed, denoted $\mathbf{t} = \mathbf{u} \otimes \mathbf{v}$. The pattern of `t` is the set intersection of `u` and `v`. The result `t` has the type of the output `ztype` of the `multiply` operator.

The `operator` is typically a `GrB_BinaryOp`, but the method is type-generic for this parameter. If given a monoid (`GrB_Monoid`), the additive operator of the monoid is used as the `multiply` binary operator. If given a semiring (`GrB_Semiring`), the multiply operator of the semiring is used as the `multiply` binary operator. The `multiply` operator may be a binary operator created by `GxB_BinaryOp_new_IndexOp`.

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.

12.4.2 GrB_Matrix_eWiseMult: element-wise matrix multiply

```
GrB_Info GrB_eWiseMult              // C<Mask> = accum (C, A.*B)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const <operator> multiply,      // defines '.*' for T=A.*B
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Matrix B,             // second input: matrix B
    const GrB_Descriptor desc       // descriptor for C, Mask, A, and B
) ;
```

`GrB_Matrix_eWiseMult` computes the element-wise "multiplication" of two matrices `A` and `B`, element-wise using any binary operator (not just times). The input matrices may be transposed first, according to the descriptor `desc`. They are then typecasted into the first and second inputs of the `multiply` operator. Next, a matrix `T` is computed, denoted $\mathbf{T} = \mathbf{A} \otimes \mathbf{B}$. The pattern of `T` is the set intersection of `A` and `B`. The result `T` has the type of the output `ztype` of the `multiply` operator.

The `multiply` operator is typically a `GrB_BinaryOp`, but the method is type-generic for this parameter. If given a monoid (`GrB_Monoid`), the additive operator of the monoid is used as the `multiply` binary operator. If given a semiring (`GrB_Semiring`), the multiply operator of the semiring is used as the `multiply` binary operator. The `multiply` operator may be a binary operator created by `GxB_BinaryOp_new_IndexOp`.

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.

12.5 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.

12.5.1 GrB_Vector_eWiseAdd: element-wise vector addition

```
GrB_Info GrB_eWiseAdd                  // w<mask> = accum (w, u+v)
(
    GrB_Vector w,                      // input/output vector for results
    const GrB_Vector mask,             // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,          // optional accum for z=accum(w,t)
    const <operator> add,              // defines '+' for t=u+v
    const GrB_Vector u,                // first input:  vector u
    const GrB_Vector v,                // second input: vector v
    const GrB_Descriptor desc          // descriptor for w and mask
) ;
```

GrB_Vector_eWiseAdd computes the element-wise "addition" of two vectors u and v, element-wise using any binary operator (not just plus). The vectors are not transposed via the descriptor. Entries in the intersection of u and v are first typecasted into the first and second inputs of the add operator. Next, a column vector t is computed, denoted $\mathbf{t} = \mathbf{u} \oplus \mathbf{v}$. The pattern of t is the set union of u and v. The result t has the type of the output ztype of the add operator.

The add operator is typically a GrB_BinaryOp, but the method is type-generic for this parameter. If given a monoid (GrB_Monoid), the additive operator of the monoid is used as the add binary operator. If given a semiring (GrB_Semiring), the additive operator of the monoid of the semiring is used as the add binary operator. The add operator may be a binary operator created by GxB_BinaryOp_new_IndexOp.

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.

12.5.2 GrB_Matrix_eWiseAdd: element-wise matrix addition

```
GrB_Info GrB_eWiseAdd                   // C<Mask> = accum (C, A+B)
(
    GrB_Matrix C,                       // input/output matrix for results
    const GrB_Matrix Mask,              // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,           // optional accum for Z=accum(C,T)
    const <operator> add,               // defines '+' for T=A+B
    const GrB_Matrix A,                 // first input:  matrix A
    const GrB_Matrix B,                 // second input: matrix B
    const GrB_Descriptor desc           // descriptor for C, Mask, A, and B
) ;
```

GrB_Matrix_eWiseAdd computes the element-wise "addition" of two matrices A and B, element-wise using any binary operator (not just plus). The input matrices may be transposed first, according to the descriptor desc. Entries in the intersection then typecasted into the first and second inputs of the add operator. Next, a matrix T is computed, denoted $\mathbf{T} = \mathbf{A} \oplus \mathbf{B}$. The pattern of T is the set union of A and B. The result T has the type of the output ztype of the add operator.

The add operator is typically a GrB_BinaryOp, but the method is type-generic for this parameter. If given a monoid (GrB_Monoid), the additive operator of the monoid is used as the add binary operator. If given a semiring (GrB_Semiring), the additive operator of the monoid of the semiring is used as the add binary operator. The add operator may be a binary operator created by GxB_BinaryOp_new_IndexOp.

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.

12.6 GxB_eWiseUnion: element-wise operations, set union

GxB_eWiseUnion computes a result with the same pattern GrB_eWiseAdd, namely, a set union of its two inputs. It differs in how the binary operator is applied.

Let $\oplus$ denote the binary operator to be used. The operator is applied to every entry in $\mathbf{A}$ and $\mathbf{B}$. A pair of scalars, α and β (alpha and beta in the API, respectively) define the inputs to the operator when entries are present in one matrix but not the other.

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

GxB_eWiseUnion is useful in contexts where GrB_eWiseAdd cannot be used because of the typecasting rules of GraphBLAS. In particular, suppose A and B are matrices with a user-defined type, and suppose < is a user-defined operator that compares two entries of this type and returns a Boolean value. Then C=A<B can be computed with GxB_eWiseUnion but not with GrB_eWiseAdd. In the latter, if A(i,j) is present but B(i,j) is not, then A(i,j) must typecasted to the type of C (GrB_BOOL in this case), and the assigment C(i,j) = (bool) A(i,j) would be performed. This is not possible because user-defined types cannot be typecasted to any other type.

Another advantage of GxB_eWiseUnion is its performance. For example, the MATLAB/Octave expression C=A-B computes C(i,j)=-B(i,j) when A(i,j) is not present. This cannot be done with a single call GrB_eWiseAdd, but it can be done with a single call to GxB_eWiseUnion, with the GrB_MINUS_FP64 operator, and with both alpha and beta scalars equal to zero. It is possible to compute this result with a temporary matrix, E=-B, computed with GrB_apply and GrB_AINV_FP64, followed by a call to GrB_eWiseAdd to compute C=A+E, but this is slower than a single call to GxB_eWiseUnion, and uses more memory.

12.6.1 GxB_Vector_eWiseUnion: element-wise vector addition

```
GrB_Info GxB_eWiseUnion             // w<mask> = accum (w, u+v)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_BinaryOp add,         // defines '+' for t=u+v
    const GrB_Vector u,             // first input:  vector u
    const GrB_Scalar alpha,
    const GrB_Vector v,             // second input: vector v
    const GrB_Scalar beta,
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

Identical to `GrB_Vector_eWiseAdd` except that two scalars are used to define how to compute the result when entries are present in one of the two input vectors (`u` and `v`), but not the other. Each of the two input scalars, `alpha` and `beta` must contain an entry. When computing the result `t=u+v`, if `u(i)` is present but `v(i)` is not, then `t(i)=u(i)+beta`. Likewise, if `v(i)` is present but `u(i)` is not, then `t(i)=alpha+v(i)`, where `+` denotes the binary operator, `add`. The `add` operator may be a binary operator created by `GxB_BinaryOp_new_IndexOp`.

12.6.2 GxB_Matrix_eWiseUnion: element-wise matrix addition

```
GrB_Info GxB_eWiseUnion            // C<M> = accum (C, A+B)
(
    GrB_Matrix C,                  // input/output matrix for results
    const GrB_Matrix Mask,         // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,      // optional accum for Z=accum(C,T)
    const GrB_BinaryOp add,        // defines '+' for T=A+B
    const GrB_Matrix A,            // first input:  matrix A
    const GrB_Scalar alpha,
    const GrB_Matrix B,            // second input: matrix B
    const GrB_Scalar beta,
    const GrB_Descriptor desc      // descriptor for C, M, A, and B
) ;
```

Identical to `GrB_Matrix_eWiseAdd` except that two scalars are used to define how to compute the result when entries are present in one of the two input matrices (`A` and `B`), but not the other. Each of the two input scalars, `alpha` and `beta` must contain an entry. When computing the result `T=A+B`, if `A(i,j)` is present but `B(i,j))` is not, then `T(i,j)=A(i,j)+beta`. Likewise, if `B(i,j)` is present but `A(i,j)` is not, then `T(i,j)=alpha+B(i,j)`, where `+` denotes the binary operator, `add`. The `add` operator may be a binary operator created by `GxB_BinaryOp_new_IndexOp`.

12.7 GrB_extract: submatrix extraction

The `GrB_extract` function is a generic name for three specific functions: `GrB_Vector_extract`, `GrB_Col_extract`, and `GrB_Matrix_extract`. The generic name appears in the function signature, but the specific function name is used when describing what each variation does.

12.7.1 GrB_Vector_extract: extract subvector from vector

```
GrB_Info GrB_extract                // w<mask> = accum (w, u(I))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_Vector u,             // first input:  vector u
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_extract` extracts a subvector from another vector, identical to `t = u (I)` in MATLAB where `I` is an integer vector of row indices. Refer to `GrB_Matrix_extract` for further details; vector extraction is the same as matrix extraction with `n`-by-1 matrices. See Section 11 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.

12.7.2 GrB_Matrix_extract: extract submatrix from matrix

```
GrB_Info GrB_extract                // C<Mask> = accum (C, A(I,J))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C, Mask, and A
) ;
```

`GrB_Matrix_extract` extracts a submatrix from another matrix, identical to `T = A(I,J)` in MATLAB where `I` and `J` are integer vectors of row and column indices, respectively, except that indices are zero-based in GraphBLAS and one-based in MATLAB. The input matrix `A` may be transposed first, via the descriptor. The type of `T` and `A` are the same. The size of `C` is `|I|`-by-`|J|`. Entries outside `A(I,J)` are not accessed and do not take part in the computation. More precisely, assuming the matrix `A` is not transposed, the matrix `T` is defined as follows:

```
T.matrix  = zeros (ni, nj) ;    % a matrix of size ni-by-nj
T.pattern = false (ni, nj) ;
for i = 1:ni
    for j = 1:nj
        if (A (I(i),J(j)).pattern)
            T (i,j).matrix  = A (I(i),J(j)).matrix ;
            T (i,j).pattern = true ;
        end
    end
end
```

If duplicate indices are present in `I` or `J`, the above method defines the result in `T`. Duplicates result in the same values of `A` being copied into different places in `T`. See Section 11 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 `GrB_ROWMAJOR`; if `|J|` is small, then it is fastest if `A` is `GrB_COLMAJOR`. The opposite is true if `A` is transposed.

12.7.3 GrB_Col_extract: extract column vector from matrix

```
GrB_Info GrB_extract                    // w<mask> = accum (w, A(I,j))
(
    GrB_Vector w,                       // input/output matrix for results
    const GrB_Vector mask,              // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,           // optional accum for z=accum(w,t)
    const GrB_Matrix A,                 // first input:  matrix A
    const GrB_Index *I,                 // row indices
    const GrB_Index ni,                 // number of row indices
    const GrB_Index j,                  // column index
    const GrB_Descriptor desc           // descriptor for w, mask, and A
) ;
```

`GrB_Col_extract` extracts a subvector from a matrix, identical to `t = A (I,j)` in MATLAB where `I` is an integer vector of row indices and where `j` is a single column index. The input matrix `A` may be transposed first, via the descriptor, which results in the extraction of a single row `j` from the matrix `A`, the result of which is a column vector `w`. The type of `t` and `A` are the same. The size of `w` is `|I|`-by-1.

See Section 11 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 `GrB_COLMAJOR`. The opposite is true if `A` is transposed.

12.8 GxB_subassign: submatrix assignment

The methods described in this section are all variations of the form `C(I,J)=A`, which modifies a submatrix of the matrix `C`. All methods can be used in their generic form with the single name `GxB_subassign`. This is reflected in the prototypes. However, to avoid confusion between the different kinds of assignment, the name of the specific function is used when describing each variation. If the discussion applies to all variations, the simple name `GxB_subassign` is used.

See Section 11 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 12.9. The two operations are compared and contrasted in Section 12.11. For a discussion of how duplicate indices are handled in `I` and `J`, see Section 12.10.

12.8.1 GxB_Vector_subassign: assign to a subvector

```
GrB_Info GxB_subassign                  // w(I)<mask> = accum (w(I),u)
(
    GrB_Vector w,                       // input/output matrix for results
    const GrB_Vector mask,              // optional mask for w(I), unused if NULL
    const GrB_BinaryOp accum,           // optional accum for z=accum(w(I),t)
    const GrB_Vector u,                 // first input:  vector u
    const GrB_Index *I,                 // row indices
    const GrB_Index ni,                 // number of row indices
    const GrB_Descriptor desc           // descriptor for w(I) and mask
) ;
```

`GxB_Vector_subassign` operates on a subvector `w(I)` of `w`, modifying it with the vector `u`. The method is identical to `GxB_Matrix_subassign` described in Section 12.8.2, where all matrices have a single column each. The `mask` has the same size as `w(I)` and `u`. The only other difference is that the input `u` in this method is not transposed via the `GrB_INP0` descriptor.

12.8.2 GxB_Matrix_subassign: assign to a submatrix

```
GrB_Info GxB_subassign              // C(I,J)<Mask> = accum (C(I,J),A)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C(I,J), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C(I,J),T)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C(I,J), Mask, and A
) ;
```

`GxB_Matrix_subassign` operates only on a submatrix `S` of `C`, modifying it with the matrix `A`. For this operation, the result is not the entire matrix `C`, but a submatrix `S=C(I,J)` of `C`. The steps taken are as follows, except that $\mathbf{A}$ may be optionally transposed via the `GrB_INP0` descriptor option.

Step	GraphBLAS notation	description
1	$\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$	extract the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix
2	$\mathbf{S}\langle\mathbf{M}\rangle = \mathbf{S} \odot \mathbf{A}$	apply the accumulator/mask to the submatrix $\mathbf{S}$
3	$\mathbf{C}(\mathbf{I}, \mathbf{J}) = \mathbf{S}$	put the submatrix $\mathbf{S}$ back into $\mathbf{C}(\mathbf{I}, \mathbf{J})$

The accumulator/mask step in Step 2 is the same as for all other GraphBLAS operations, described in Section 2.3, except that for `GxB_subassign`, it is applied to just the submatrix $\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$, and thus the `Mask` has the same size as $\mathbf{A}$, $\mathbf{S}$, and $\mathbf{C}(\mathbf{I}, \mathbf{J})$.

The `GxB_subassign` operation is the reverse of matrix extraction:

- For submatrix extraction, `GrB_Matrix_extract`, the submatrix `A(I,J)` appears on the right-hand side of the assignment, `C=A(I,J)`, and entries outside of the submatrix are not accessed and do not take part in the computation.

- For submatrix assignment, `GxB_Matrix_subassign`, the submatrix `C(I,J)` appears on the left-hand-side of the assignment, `C(I,J)=A`, and entries outside of the submatrix are not accessed and do not take part in the computation.

In both methods, the accumulator and mask modify the submatrix of the assignment; they simply differ on which side of the assignment the submatrix resides on. In both cases, if the Mask matrix is present it is the same size as the submatrix:

- For submatrix extraction, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}(\mathbf{I},\mathbf{J})$ is computed, where the submatrix is on the right. The mask $\mathbf{M}$ has the same size as the submatrix $\mathbf{A}(\mathbf{I},\mathbf{J})$.

- For submatrix assignment, $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{I},\mathbf{J}) \odot \mathbf{A}$ is computed, where the submatrix is on the left. The mask $\mathbf{M}$ has the same size as the submatrix $\mathbf{C}(\mathbf{I},\mathbf{J})$.

In Step 1, the submatrix S is first computed by the GrB_Matrix_extract operation, S=C(I,J).

Step 2 accumulates the results $\mathbf{S}\langle\mathbf{M}\rangle = \mathbf{S} \odot \mathbf{T}$, exactly as described in Section 2.3, but operating on the submatrix $\mathbf{S}$, not $\mathbf{C}$, using the optional Mask and accum operator. The matrix $\mathbf{T}$ is simply $\mathbf{T} = \mathbf{A}$, or $\mathbf{T} = \mathbf{A}^{\mathsf{T}}$ if $\mathbf{A}$ is transposed via the desc descriptor, GrB_INP0. The GrB_REPLACE option in the descriptor clears $\mathbf{S}$ after computing $\mathbf{Z} = \mathbf{T}$ or $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$, not all of $\mathbf{C}$ since this operation can only modify the specified submatrix of $\mathbf{C}$.

Finally, Step 3 writes the result (which is the modified submatrix S and not all of C) back into the C matrix that contains it, via the assignment C(I,J)=S, using the reverse operation from the method described for matrix extraction:

```
for i = 1:ni
    for j = 1:nj
        if (S (i,j).pattern)
            C (I(i),J(j)).matrix = S (i,j).matrix ;
            C (I(i),J(j)).pattern = true ;
        end
    end
end
```

Performance considerations: If A is not transposed: if |I| is small, then it is fastest if the format of C is GrB_ROWMAJOR; if |J| is small, then it is fastest if the format of C is GrB_COLMAJOR. The opposite is true if A is transposed.

12.8.3 GxB_Col_subassign: assign to a sub-column of a matrix

```
GrB_Info GxB_subassign              // C(I,j)<mask> = accum (C(I,j),u)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(I,j), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(I,j),t)
    const GrB_Vector u,             // input vector
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index j,              // column index
    const GrB_Descriptor desc       // descriptor for C(I,j) and mask
) ;
```

`GxB_Col_subassign` modifies a single sub-column of a matrix `C`. It is the same as `GxB_Matrix_subassign` where the index vector `J[0]=j` is a single column index (and thus `nj=1`), and where all matrices in `GxB_Matrix_subassign` (except `C`) consist of a single column. The `mask` vector has the same size as `u` and the sub-column `C(I,j)`. The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GxB_Matrix_subassign` for further details.

Performance considerations: `GxB_Col_subassign` is much faster than `GxB_Row_subassign` if the format of `C` is `GrB_COLMAJOR`. `GxB_Row_subassign` is much faster than `GxB_Col_subassign` if the format of `C` is `GrB_ROWMAJOR`.

12.8.4 GxB_Row_subassign: assign to a sub-row of a matrix

```
GrB_Info GxB_subassign              // C(i,J)<mask'> = accum (C(i,J),u')
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(i,J), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(i,J),t)
    const GrB_Vector u,             // input vector
    const GrB_Index i,              // row index
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C(i,J) and mask
) ;
```

`GxB_Row_subassign` modifies a single sub-row of a matrix `C`. It is the same as `GxB_Matrix_subassign` where the index vector `I[0]=i` is a single row index (and thus `ni=1`), and where all matrices in `GxB_Matrix_subassign` (except `C`) consist of a single row. The `mask` vector has the same size as `u` and the sub-column `C(I,j)`. The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GxB_Matrix_subassign` for further details.

Performance considerations: `GxB_Col_subassign` is much faster than `GxB_Row_subassign` if the format of `C` is `GrB_COLMAJOR`. `GxB_Row_subassign` is much faster than `GxB_Col_subassign` if the format of `C` is `GrB_ROWMAJOR`.

12.8.5 GxB_Vector_subassign_<type>: assign a scalar to a subvector

```
GrB_Info GxB_subassign                  // w(I)<mask> = accum (w(I),x)
(
    GrB_Vector w,                       // input/output vector for results
    const GrB_Vector mask,              // optional mask for w(I), unused if NULL
    const GrB_BinaryOp accum,           // optional accum for z=accum(w(I),x)
    const <type> x,                     // scalar to assign to w(I)
    const GrB_Index *I,                 // row indices
    const GrB_Index ni,                 // number of row indices
    const GrB_Descriptor desc           // descriptor for w(I) and mask
) ;
```

`GxB_Vector_subassign_<type>` assigns a single scalar to an entire subvector of the vector `w`. The operation is exactly like setting a single entry in an `n`-by-1 matrix, `A(I,0) = x`, where the column index for a vector is implicitly `j=0`. For further details of this function, see `GxB_Matrix_subassign_<type>` in Section 12.8.6.

12.8.6 GxB_Matrix_subassign_<type>: assign a scalar to a submatrix

```
GrB_Info GxB_subassign              // C(I,J)<Mask> = accum (C(I,J),x)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C(I,J), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C(I,J),x)
    const <type> x,                 // scalar to assign to C(I,J)
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C(I,J) and Mask
) ;
```

`GxB_Matrix_subassign_<type>` assigns a single scalar to an entire submatrix of `C`, like the *scalar expansion* `C(I,J)=x` in MATLAB. The scalar `x` is implicitly expanded into a matrix `A` of size `ni` by `nj`, with all entries present and equal to `x`, and then the matrix `A` is assigned to `C(I,J)` using the same method as in `GxB_Matrix_subassign`. Refer to that function in Section 12.8.2 for further details. For the accumulation step, the scalar `x` is typecasted directly into the type of `C` when the `accum` operator is not applied to it, or into the `ytype` of the `accum` operator, if `accum` is not NULL, for entries that are already present in `C`.

The `<type> x` notation is otherwise the same as `GrB_Matrix_setElement` (see Section 6.10.10). Any value can be passed to this function and its type will be detected, via the `_Generic` feature of C11. For a user-defined type, `x` is a `void *` pointer that points to a memory space holding a single entry of a scalar that has exactly the same user-defined type as the matrix `C`. This user-defined type must exactly match the user-defined type of `C` since no typecasting is done between user-defined types.

If a `void *` pointer is passed in and the type of the underlying scalar does not exactly match the user-defined type of `C`, then results are undefined. No error status will be returned since GraphBLAS has no way of catching this error. If `x` is a `GrB_Scalar` with no entry, then it is implicitly expanded into a matrix `A` of size `ni` by `nj`, with no entries present.

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GrB_ROWMAJOR`; if `|J|` is small, then it is fastest if the format of `C` is `GrB_COLMAJOR`. The opposite is true if `A` is transposed.

12.9 GrB_assign: submatrix assignment

The methods described in this section are all variations of the form `C(I,J)=A`, which modifies a submatrix of the matrix `C`. All methods can be used in their generic form with the single name `GrB_assign`. These methods are very similar to their `GxB_subassign` counterparts in Section 12.8. They differ primarily in the size of the `Mask`, and how the `GrB_REPLACE` option works. Section 12.11 compares `GxB_subassign` and `GrB_assign`.

See Section 11 for a description of `I`, `ni`, `J`, and `nj`.

12.9.1 GrB_Vector_assign: assign to a subvector

```
GrB_Info GrB_assign                 // w<mask>(I) = accum (w(I),u)
(
    GrB_Vector w,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w(I),t)
    const GrB_Vector u,             // first input:  vector u
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_assign` operates on a subvector `w(I)` of `w`, modifying it with the vector `u`. The `mask` vector has the same size as `w`. The method is identical to `GrB_Matrix_assign` described in Section 12.9.2, where all matrices have a single column each. The only other difference is that the input `u` in this method is not transposed via the `GrB_INP0` descriptor.

12.9.2 GrB_Matrix_assign: assign to a submatrix

```
GrB_Info GrB_assign                 // C<Mask>(I,J) = accum (C(I,J),A)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C(I,J),T)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C, Mask, and A
) ;
```

`GrB_Matrix_assign` operates on a submatrix `S` of `C`, modifying it with the matrix `A`. It may also modify all of `C`, depending on the input descriptor `desc` and the `Mask`.

Step	GraphBLAS notation	description
1	$\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$	extract $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix
2	$\mathbf{S} = \mathbf{S} \odot \mathbf{A}$	apply the accumulator (but not the mask) to $\mathbf{S}$
3	$\mathbf{Z} = \mathbf{C}$	make a copy of $\mathbf{C}$
4	$\mathbf{Z}(\mathbf{I}, \mathbf{J}) = \mathbf{S}$	put the submatrix into $\mathbf{Z}(\mathbf{I}, \mathbf{J})$
5	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$	apply the mask/replace phase to all of $\mathbf{C}$

In contrast to `GxB_subassign`, the `Mask` has the same as `C`.

Step 1 extracts the submatrix and then Step 2 applies the accumulator (or $\mathbf{S} = \mathbf{A}$ if `accum` is `NULL`). The `Mask` is not yet applied.

Step 3 makes a copy of the $\mathbf{C}$ matrix, and then Step 4 writes the submatrix $\mathbf{S}$ into $\mathbf{Z}$. This is the same as Step 3 of `GxB_subassign`, except that it operates on a temporary matrix $\mathbf{Z}$.

Finally, Step 5 writes $\mathbf{Z}$ back into $\mathbf{C}$ via the `Mask`, using the Mask/Replace Phase described in Section 2.3. If `GrB_REPLACE` is enabled, then all of $\mathbf{C}$ is cleared prior to writing $\mathbf{Z}$ via the mask. As a result, the `GrB_REPLACE` option can delete entries outside the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix.

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GrB_ROWMAJOR`; if `|J|` is small, then it is fastest if the format of `C` is `GrB_COLMAJOR`. The opposite is true if `A` is transposed.

12.9.3 GrB_Col_assign: assign to a sub-column of a matrix

```
GrB_Info GrB_assign                 // C<mask>(I,j) = accum (C(I,j),u)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(:,j), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(I,j),t)
    const GrB_Vector u,             // input vector
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index j,              // column index
    const GrB_Descriptor desc       // descriptor for C(:,j) and mask
) ;
```

`GrB_Col_assign` modifies a single sub-column of a matrix `C`. It is the same as `GrB_Matrix_assign` where the index vector `J[0]=j` is a single column index, and where all matrices in `GrB_Matrix_assign` (except `C`) consist of a single column.

Unlike `GrB_Matrix_assign`, the `mask` is a vector with the same size as a single column of `C`.

The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GrB_Matrix_assign` for further details.

Performance considerations: `GrB_Col_assign` is much faster than `GrB_Row_assign` if the format of `C` is `GrB_COLMAJOR`. `GrB_Row_assign` is much faster than `GrB_Col_assign` if the format of `C` is `GrB_ROWMAJOR`.

12.9.4 GrB_Row_assign: assign to a sub-row of a matrix

```
GrB_Info GrB_assign                 // C<mask'>(i,J) = accum (C(i,J),u')
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(i,:), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(i,J),t)
    const GrB_Vector u,             // input vector
    const GrB_Index i,              // row index
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C(i,:) and mask
) ;
```

`GrB_Row_assign` modifies a single sub-row of a matrix `C`. It is the same as `GrB_Matrix_assign` where the index vector `I[0]=i` is a single row index, and where all matrices in `GrB_Matrix_assign` (except `C`) consist of a single row.

Unlike `GrB_Matrix_assign`, the `mask` is a vector with the same size as a single row of `C`.

The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GrB_Matrix_assign` for further details.

Performance considerations: `GrB_Col_assign` is much faster than `GrB_Row_assign` if the format of `C` is `GrB_COLMAJOR`. `GrB_Row_assign` is much faster than `GrB_Col_assign` if the format of `C` is `GrB_ROWMAJOR`.

12.9.5 GrB_Vector_assign_<type>: assign a scalar to a subvector

```
GrB_Info GrB_assign                 // w<mask>(I) = accum (w(I),x)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w(I),x)
    const <type> x,                 // scalar to assign to w(I)
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_assign_<type>` assigns a single scalar to an entire subvector of the vector `w`. The operation is exactly like setting a single entry in an `n`-by-1 matrix, `A(I,0) = x`, where the column index for a vector is implicitly `j=0`. The `mask` vector has the same size as `w`. For further details of this function, see `GrB_Matrix_assign_<type>` in the next section (12.9.6).

Following the C API Specification, results are well-defined if `I` contains duplicate indices. Duplicate indices are simply ignored. See Section 12.10 for more details.

12.9.6 GrB_Matrix_assign_<type>: assign a scalar to a submatrix

```
GrB_Info GrB_assign                 // C<Mask>(I,J) = accum (C(I,J),x)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C(I,J),x)
    const <type> x,                 // scalar to assign to C(I,J)
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C and Mask
) ;
```

`GrB_Matrix_assign_<type>` assigns a single scalar to an entire submatrix of `C`, like the *scalar expansion* `C(I,J)=x` in MATLAB. The scalar `x` is implicitly expanded into a matrix `A` of size `ni` by `nj`, and then the matrix `A` is assigned to `C(I,J)` using the same method as in `GrB_Matrix_assign`. Refer to that function in Section 12.9.2 for further details.

The `Mask` has the same size as `C`.

For the accumulation step, the scalar `x` is typecasted directly into the type of `C` when the `accum` operator is not applied to it, or into the `ytype` of the `accum` operator, if `accum` is not NULL, for entries that are already present in `C`.

The `<type> x` notation is otherwise the same as `GrB_Matrix_setElement` (see Section 6.10.10). Any value can be passed to this function and its type will be detected, via the `_Generic` feature of C11. For a user-defined type, `x` is a `void *` pointer that points to a memory space holding a single entry of a scalar that has exactly the same user-defined type as the matrix `C`. This user-defined type must exactly match the user-defined type of `C` since no typecasting is done between user-defined types.

If a `void *` pointer is passed in and the type of the underlying scalar does not exactly match the user-defined type of `C`, then results are undefined. No error status will be returned since GraphBLAS has no way of catching this error.

If `x` is a `GrB_Scalar` with no entry, then it is implicitly expanded into a matrix `A` of size `ni` by `nj`, with no entries present.

Following the C API Specification, results are well-defined if `I` or `J` contain duplicate indices. Duplicate indices are simply ignored. See Section 12.10 for more details.

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GrB_ROWMAJOR`; if `|J|` is small, then it is fastest if the format of `C` is `GrB_COLMAJOR`. The opposite is true if `A` is transposed.

12.10 Duplicate indices in GrB_assign and GxB_subassign

According to the GraphBLAS C API Specification if the index vectors I or J contain duplicate indices, the results are undefined for GrB_Matrix_assign, GrB_Matrix_assign, GrB_Col_assign, and GrB_Row_assign. Only the scalar assignment operations (GrB_Matrix_assign_TYPE and GrB_Matrix_assign_TYPE) are well-defined when duplicates appear in I and J. In those two functions, duplicate indices are ignored.

As an extension to the specification, SuiteSparse:GraphBLAS provides a definition of how duplicate indices are handled in all cases. If I has duplicate indices, they are ignored and the last unique entry in the list is used. When no mask and no accumulator is present, the results are identical to how MATLAB handles duplicate indices in the built-in expression C(I,J)=A. Details of how this is done is shown below.

```
function C = subassign (C, I, J, A)
% submatrix assignment with pre-sort of I and J; and remove duplicates

% delete duplicates from I, keeping the last one seen
[I2 I2k] = sort (I) ;
Idupl = [(I2 (1:end-1) == I2 (2:end)), false] ;
I2  = I2  (~Idupl) ;
I2k = I2k (~Idupl) ;
assert (isequal (I2, unique (I)))

% delete duplicates from J, keeping the last one seen
[J2 J2k] = sort (J) ;
Jdupl = [(J2 (1:end-1) == J2 (2:end)), false] ;
J2  = J2  (~Jdupl) ;
J2k = J2k (~Jdupl) ;
assert (isequal (J2, unique (J)))

% do the submatrix assignment, with no duplicates in I2 or J2
C (I2,J2) = A (I2k,J2k) ;
```

If a mask is present, then it is replaced with M = M (I2k, J2k) for GxB_subassign, or with M = M (I2, J2) for GrB_assign. If an accumulator operator is present, it is applied after the duplicates are removed, as (for example):

```
C (I2,J2) = C (I2,J2) + A (I2k,J2k) ;
```

These definitions allow the MATLAB/Octave interface to GraphBLAS to return the same results for `C(I,J)=A` for a `GrB` object as they do for built-in MATLAB/Octave 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"
#undef I    /* complex.h #define's I, but I is used an array below */
#include <stdio.h>
int main (void)
{
    GrB_init (GrB_NONBLOCKING) ;
    GrB_Vector x, y ;
    GrB_Vector_new (&x, GrB_INT32, 1) ;
    GrB_Vector_new (&y, GrB_INT32, 2) ;
    GrB_Index I [2] = {0, 0} ;
    GrB_Vector_setElement (y, 5, 0) ;
    GrB_Vector_setElement (y, 7, 1) ;
    GrB_Vector_wait (&y) ;
    GxB_print (x, 3) ;
    GxB_print (y, 3) ;
    GrB_assign (x, NULL, GrB_MIN_INT32, y, I, 2, NULL) ;
    GrB_Vector_wait (&y) ;
    GxB_print (x, 3) ;
    GrB_finalize ( ) ;
}
```

You might (wrongly) expect the result to be the vector `x(0)=5`, since two entries seem to be assigned, and the min operator might be expected to take the minimum of the two. This is not how SuiteSparse:GraphBLAS handles duplicates.

Instead, the first duplicate index of `I` is discarded (`I [0] = 0`, and `y(0)=5`). and only the second entry is used (`I [1] = 0`, and `y(1)=7`). The output of the above program is:

```
1x1 GraphBLAS int32_t vector, sparse by col:
x, no entries


2x1 GraphBLAS int32_t vector, sparse by col:
y, 2 entries

  (0,0)   5
  (1,0)   7


1x1 GraphBLAS int32_t vector, sparse by col:
x, 1 entry

  (0,0)   7
```

You see that the result is `x(0)=7`, since the `y(0)=5` entry has been ignored because of the duplicate indices in `I`.

SPEC: Providing a well-defined behavior for duplicate indices with matrix and vector assignment is an extension to the specification. The specification only defines the behavior when assigning a scalar into a matrix or vector, and states that duplicate indices otherwise lead to undefined results.

12.11 Comparing GrB_assign and GxB_subassign

The GxB_subassign and GrB_assign operations are very similar, but they differ in two ways:

1. **The Mask has a different size:** The mask in GxB_subassign has the same dimensions as w(I) for vectors and C(I,J) for matrices. In GrB_assign, the mask is the same size as w or C, respectively (except for the row/col variants). The two masks are related. If M is the mask for GrB_assign, then M(I,J) is the mask for GxB_subassign. If there is no mask, or if I and J are both GrB_ALL, the two masks are the same. For GrB_Row_assign and GrB_Col_assign, the mask vector is the same size as a row or column of C, respectively. For the corresponding GxB_Row_subassign and GxB_Col_subassign operations, the mask is the same size as the sub-row C(i,J) or subcolumn C(I,j), respectively.

2. GrB_REPLACE **is different:** They differ in how C is affected in areas outside the C(I,J) submatrix. In GxB_subassign, the C(I,J) submatrix is the only part of C that can be modified, and no part of C outside the submatrix is ever modified. In GrB_assign, it is possible to delete entries in C outside the submatrix, but only in one specific manner. Suppose the mask M is present (or, suppose it is not present but GrB_COMP is true). After (optionally) complementing the mask, the value of M(i,j) can be 0 for some entry outside the C(I,J) submatrix. If the GrB_REPLACE descriptor is true, GrB_assign deletes this entry.

GxB_subassign and GrB_assign are identical if GrB_REPLACE is set to its default value of false, and if the masks happen to be the same. The two masks can be the same in two cases: either the Mask input is NULL (and it is not complemented via GrB_COMP), or I and J are both GrB_ALL. If all these conditions hold, the two algorithms are identical and have the same performance. Otherwise, GxB_subassign is much faster than GrB_assign when the latter must examine the entire matrix C to delete entries (when GrB_REPLACE is true), and if it must deal with a much larger Mask matrix. However, both methods have specific uses.

Consider using C(I,J)+=F for many submatrices F (for example, when assembling a finite-element matrix). If the Mask is meant as a specification for which entries of C should appear in the final result, then use GrB_assign.

If instead the Mask is meant to control which entries of the submatrix C(I,J) are modified by the finite-element F, then use GxB_subassign. This is particularly useful is the Mask is a template that follows along with the finite-element F, independent of where it is applied to C. Using GrB_assign would be very difficult in this case since a new Mask, the same size as C, would need to be constructed for each finite-element F.

In GraphBLAS notation, the two methods can be described as follows:

matrix and vector subassign	$\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{I},\mathbf{J}) \odot \mathbf{A}$
matrix and vector assign	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I},\mathbf{J}) = \mathbf{C}(\mathbf{I},\mathbf{J}) \odot \mathbf{A}$

This notation does not include the details of the GrB_COMP and GrB_REPLACE descriptors, but it does illustrate the difference in the Mask. In the subassign, Mask is the same size as C(I,J) and A. If I[0]=i and J[0]=j, Then Mask(0,0) controls how C(i,j) is modified by the subassign, from the value A(0,0). In the assign, Mask is the same size as C, and Mask(i,j) controls how C(i,j) is modified.

The GxB_subassign and GrB_assign functions have the same signatures; they differ only in how they consider the Mask and the GrB_REPLACE descriptor

Details of each step of the two operations are listed below:

Step	GrB_Matrix_assign	GxB_Matrix_subassign
1	$\mathbf{S} = \mathbf{C}(\mathbf{I},\mathbf{J})$	$\mathbf{S} = \mathbf{C}(\mathbf{I},\mathbf{J})$
2	$\mathbf{S} = \mathbf{S} \odot \mathbf{A}$	$\mathbf{S}\langle\mathbf{M}\rangle = \mathbf{S} \odot \mathbf{A}$
3	$\mathbf{Z} = \mathbf{C}$	$\mathbf{C}(\mathbf{I},\mathbf{J}) = \mathbf{S}$
4	$\mathbf{Z}(\mathbf{I},\mathbf{J}) = \mathbf{S}$	
5	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$	

Step 1 is the same. In the Accumulator Phase (Step 2), the expression $\mathbf{S} \odot \mathbf{A}$, described in Section 2.3, is the same in both operations. The result is simply $\mathbf{A}$ if accum is NULL. It only applies to the submatrix $\mathbf{S}$, not the whole matrix. The result $\mathbf{S} \odot \mathbf{A}$ is used differently in the Mask/Replace phase.

The Mask/Replace Phase, described in Section 2.3 is different:

- For GrB_assign (Step 5), the mask is applied to all of $\mathbf{C}$. The mask has the same size as $\mathbf{C}$. Just prior to making the assignment via the mask, the GrB_REPLACE option can be used to clear all of $\mathbf{C}$ first. This is the only way in which entries in $\mathbf{C}$ that are outside the $\mathbf{C}(\mathbf{I},\mathbf{J})$ submatrix can be modified by this operation.

- For `GxB_subassign` (Step 2b), the mask is applied to just $\mathbf{S}$. The mask has the same size as $\mathbf{C}(\mathbf{I},\mathbf{J})$, $\mathbf{S}$, and $\mathbf{A}$. Just prior to making the assignment via the mask, the `GrB_REPLACE` option can be used to clear $\mathbf{S}$ first. No entries in $\mathbf{C}$ that are outside the $\mathbf{C}(\mathbf{I},\mathbf{J})$ can be modified by this operation. Thus, `GrB_REPLACE` has no effect on entries in $\mathbf{C}$ outside the $\mathbf{C}(\mathbf{I},\mathbf{J})$ submatrix.

The differences between `GrB_assign` and `GxB_subassign` can be seen in Tables 5 and 6. The first table considers the case when the entry c_{ij} is in the $\mathbf{C}(\mathbf{I},\mathbf{J})$ submatrix, and it describes what is computed for both `GrB_assign` and `GxB_subassign`. They perform the exact same computation; the only difference is how the value of the mask is specified. Compare Table 5 with Table 1 in Section 7.

The first column of Table 5 is *yes* if `GrB_REPLACE` is enabled, and a dash otherwise. The second column is *yes* if an accumulator operator is given, and a dash otherwise. The third column is c_{ij} if the entry is present in $\mathbf{C}$, and a dash otherwise. The fourth column is $a_{i'j'}$ if the corresponding entry is present in $\mathbf{A}$, where $i = \mathbf{I}(i')$ and $j = \mathbf{J}(i')$.

The *mask* column is 1 if the effective value of the mask mask allows $\mathbf{C}$ to be modified, and 0 otherwise. This is m_{ij} for `GrB_assign`, and $m_{i'j'}$ for `GxB_subassign`, to reflect the difference in the mask, but this difference is not reflected in the table. The value 1 or 0 is the value of the entry in the mask after it is optionally complemented via the `GrB_COMP` option.

Finally, the last column is the action taken in this case. It is left blank if no action is taken, in which case c_{ij} is not modified if present, or not inserted into $\mathbf{C}$ if not present.

repl	accum	$\mathbf{C}$	$\mathbf{A}$	mask	action taken by `GrB_assign` and `GxB_subassign`
-	-	c_{ij}	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, update
-	-	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
-	-	c_{ij}	-	1	delete c_{ij} because $a_{i'j'}$ not present
-	-	-	-	1	
-	-	c_{ij}	$a_{i'j'}$	0	
-	-	-	$a_{i'j'}$	0	
-	-	c_{ij}	-	0	
-	-	-	-	0	
yes	-	c_{ij}	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, update
yes	-	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
yes	-	c_{ij}	-	1	delete c_{ij} because $a_{i'j'}$ not present
yes	-	-	-	1	
yes	-	c_{ij}	$a_{i'j'}$	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	$a_{i'j'}$	0	
yes	-	c_{ij}	-	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	-	0	
-	yes	c_{ij}	$a_{i'j'}$	1	$c_{ij} = c_{ij} \odot a_{i'j'}$, apply accumulator
-	yes	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
-	yes	c_{ij}	-	1	
-	yes	-	-	1	
-	yes	c_{ij}	$a_{i'j'}$	0	
-	yes	-	$a_{i'j'}$	0	
-	yes	c_{ij}	-	0	
-	yes	-	-	0	
yes	yes	c_{ij}	$a_{i'j'}$	1	$c_{ij} = c_{ij} \odot a_{i'j'}$, apply accumulator
yes	yes	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
yes	yes	c_{ij}	-	1	
yes	yes	-	-	1	
yes	yes	c_{ij}	$a_{i'j'}$	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	$a_{i'j'}$	0	
yes	yes	c_{ij}	-	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	-	0	

Table 5: Results of assign and subassign for entries in the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix

repl	accum	$\mathbf{C}$	$\mathbf{C}=\mathbf{Z}$	mask	action taken by `GrB_assign`
-	-	c_{ij}	c_{ij}	1	
-	-	-	-	1	
-	-	c_{ij}	c_{ij}	0	
-	-	-	-	0	
yes	-	c_{ij}	c_{ij}	1	
yes	-	-	-	1	
yes	-	c_{ij}	c_{ij}	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	-	0	
-	yes	c_{ij}	c_{ij}	1	
-	yes	-	-	1	
-	yes	c_{ij}	c_{ij}	0	
-	yes	-	-	0	
yes	yes	c_{ij}	c_{ij}	1	
yes	yes	-	-	1	
yes	yes	c_{ij}	c_{ij}	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	-	0	

Table 6: Results of assign for entries outside the $\mathbf{C}(\mathbf{I},\mathbf{J})$ submatrix. Subassign has no effect on these entries.

Table 6 illustrates how `GrB_assign` and `GxB_subassign` differ for entries outside the submatrix. `GxB_subassign` never modifies any entry outside the $\mathbf{C}(\mathbf{I},\mathbf{J})$ submatrix, but `GrB_assign` can modify them in two cases listed in Table 6. When the `GrB_REPLACE` option is selected, and when the `Mask(i,j)` for an entry c_{ij} is false (or if the `Mask(i,j)` is true and `GrB_COMP` is enabled via the descriptor), then the entry is deleted by `GrB_assign`.

The fourth column of Table 6 differs from Table 5, since entries in $\mathbf{A}$ never affect these entries. Instead, for all index pairs outside the $I \times J$ submatrix, $\mathbf{C}$ and $\mathbf{Z}$ are identical (see Step 3 above). As a result, each section of the table includes just two cases: either c_{ij} is present, or not. This in contrast to Table 5, where each section must consider four different cases.

The `GrB_Row_assign` and `GrB_Col_assign` operations are slightly different. They only affect a single row or column of $\mathbf{C}$. For `GrB_Row_assign`, Table 6 only applies to entries in the single row `C(i,J)` that are outside the list of indices, `J`. For `GrB_Col_assign`, Table 6 only applies to entries in the single column `C(I,j)` that are outside the list of indices, `I`.

12.11.1 Example

The difference between `GxB_subassign` and `GrB_assign` is illustrated in the following example. Consider the 2-by-2 matrix $\mathbf{C}$ where all entries are present.

$$\mathbf{C} = \left[\begin{array}{rr} 11 & 12 \\ 21 & 22 \end{array} \right]$$

Suppose `GrB_REPLACE` is true, and `GrB_COMP` is false. Let the `Mask` be:

$$\mathbf{M} = \left[\begin{array}{rr} 1 & 1 \\ 0 & 1 \end{array} \right].$$

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:

$$\mathbf{C} = \left[\begin{array}{rr} 11 & 112 \\ - & 22 \end{array} \right].$$

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:

$$\mathbf{M} = \left[\begin{array}{r} 1 \end{array} \right].$$

After computing $\mathbf{C}(0,1)\langle \mathbf{M} \rangle = \mathbf{C}(0,1) + \mathbf{A}$, the result is

$$\mathbf{C} = \left[\begin{array}{rr} 11 & 112 \\ 21 & 22 \end{array} \right].$$

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`.

12.11.2 Performance of GxB_subassign, GrB_assign and GrB_*_setElement

When SuiteSparse:GraphBLAS uses non-blocking mode, the modifications to a matrix by `GxB_subassign`, `GrB_assign`, and `GrB_*_setElement` can postponed, and computed all at once later on. This has a huge impact on performance.

A sequence of assignments is fast if their completion can be postponed for as long as possible, or if they do not modify the pattern at all. Modifying the pattern can be costly, but it is fast if non-blocking mode can be fully exploited.

Consider a sequence of t submatrix assignments `C(I,J)=C(I,J)+A` to an n-by-n matrix `C` where each submatrix `A` has size a-by-a with s entries, and where `C` starts with c entries. Assume the matrices are all stored in non-hypersparse form, by row (`GrB_ROWMAJOR`).

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 12.11. It can be up to 1000x faster than MATLAB R2019b, or even higher depending on the kind of matrix assignment. MATLAB logical indexing (the mask of GraphBLAS) is extremely faster with GraphBLAS as compared in MATLAB R2019b; differences of up to 250,000x have been observed (0.4 seconds in GraphBLAS versus 28 hours in MATLAB).

All of the 28 variants (each with their own source code) are either asymptotically optimal, or to within a log factor of being asymptotically optimal. The methods are also fully parallel. For hypersparse matrices, the term n in the expressions in the above discussion is dropped, and is replaced with $h \log h$, at the worst case, where $h << n$ is the number of non-empty columns of a hypersparse matrix stored by column, or the number of non-empty rows of a hypersparse matrix stored by row. In many methods, n is replaced with h, not $h \log h$.

12.12 GrB_apply: apply a unary, binary, or index-unary operator

GrB_apply is the generic name for 92 specific functions:

- GrB_Vector_apply and GrB_Matrix_apply apply a unary operator to the entries of a matrix (two variants).
- GrB_*_apply_BinaryOp1st_* applies a binary operator where a single scalar is provided as the x input to the binary operator. There are 30 variants, depending on the type of the scalar: (matrix or vector) x (13 built-in types, one for user-defined types, and a version for GrB_Scalar).
- GrB_*_apply_BinaryOp2nd_* applies a binary operator where a single scalar is provided as the y input to the binary operator. There are 30 variants, depending on the type of the scalar: (matrix or vector) x (13 built-in types, one for user-defined types, and a version for GrB_Scalar).
- GrB_*_apply_IndexOp_* applies a GrB_IndexUnaryOp, single scalar is provided as the scalar y input to the index-unary operator. There are 30 variants, depending on the type of the scalar: (matrix or vector) x (13 built-in types, one for user-defined types, and a version for GrB_Scalar).

The generic name appears in the function prototypes, but the specific function name is used when describing each variation. When discussing features that apply to all versions, the simple name GrB_apply is used.

12.12.1 GrB_Vector_apply: apply a unary operator to a vector

```
GrB_Info GrB_apply                  // w<mask> = accum (w, op(u))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_UnaryOp op,           // operator to apply to the entries
    const GrB_Vector u,             // first input:  vector u
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

GrB_Vector_apply applies a unary operator to the entries of a vector, analogous to t = op(u) in MATLAB except the operator op is only applied to entries in the pattern of `u`. Implicit values outside the pattern of `u` are not affected. The entries in `u` are typecasted into the `xtype` of the unary operator. The vector `t` has the same type as the `ztype` of the unary operator. The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

12.12.2 GrB_Matrix_apply: apply a unary operator to a matrix

```
GrB_Info GrB_apply                  // C<Mask> = accum (C, op(A)) or op(A')
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_UnaryOp op,           // operator to apply to the entries
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

`GrB_Matrix_apply` applies a unary operator to the entries of a matrix, analogous to `T = op(A)` in MATLAB except the operator `op` is only applied to entries in the pattern of `A`. Implicit values outside the pattern of `A` are not affected. The input matrix `A` may be transposed first. The entries in `A` are typecasted into the `xtype` of the unary operator. The matrix `T` has the same type as the `ztype` of the unary operator. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

The built-in `GrB_IDENTITY_`T operators (one for each built-in type T) are very useful when combined with this function, enabling it to compute $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}$. This makes `GrB_apply` a direct interface to the accumulator/mask function for both matrices and vectors. The `GrB_IDENTITY_`T operators also provide the fastest stand-alone typecasting methods in SuiteSparse:GraphBLAS, with all $13 \times 13 = 169$ methods appearing as individual functions, to typecast between any of the 13 built-in types.

To compute $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}$ or $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}$ for user-defined types, the user application would need to define an identity operator for the type. Since GraphBLAS cannot detect that it is an identity operator, it must call the operator to make the full copy `T=A` and apply the operator to each entry of the matrix or vector.

The other GraphBLAS operation that provides a direct interface to the accumulator/mask function is `GrB_transpose`, which does not require an operator to perform this task. As a result, `GrB_transpose` can be used as an efficient and direct interface to the accumulator/mask function for both built-in and user-defined types. However, it is only available for matrices, not vectors.

12.12.3 GrB_Vector_apply_BinaryOp1st: apply a binary operator to a vector; 1st scalar binding

```
GrB_Info GrB_apply                  // w<mask> = accum (w, op(x,u))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_BinaryOp op,          // operator to apply to the entries
    <type> x,                       // first input:  scalar x
    const GrB_Vector u,             // second input: vector u
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_apply_BinaryOp1st_<type>` applies a binary operator $z = f(x,y)$ to a vector, where a scalar x is bound to the first input of the operator. The scalar `x` can be a non-opaque C scalar corresponding to a built-in type, a `void *` for user-defined types, or a `GrB_Scalar`. It is otherwise identical to `GrB_Vector_apply`.

The `op` can be any binary operator except that it cannot be a user-defined `GrB_BinaryOp` created by `GxB_BinaryOp_new_IndexOp`. For backward compatibility with prior versions of SuiteSparse:GraphBLAS, built-in index-based binary operators such as `GxB_FIRSTI_INT32` may be used, however. The equivalent index-unary operators are used in their place.

12.12.4 GrB_Vector_apply_BinaryOp2nd: apply a binary operator to a vector; 2nd scalar binding

```
GrB_Info GrB_apply                  // w<mask> = accum (w, op(u,y))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_BinaryOp op,          // operator to apply to the entries
    const GrB_Vector u,             // first input:  vector u
    <type> y,                       // second input: scalar y
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_apply_BinaryOp2nd_<type>` applies a binary operator $z = f(x, y)$ to a vector, where a scalar y is bound to the second input of the operator. The scalar `x` can be a non-opaque C scalar corresponding to a built-in type, a `void *` for user-defined types, or a `GrB_Scalar`. It is otherwise identical to `GrB_Vector_apply`.

The `op` can be any binary operator except that it cannot be a user-defined `GrB_BinaryOp` created by `GxB_BinaryOp_new_IndexOp`. For backward compatibility with prior versions of SuiteSparse:GraphBLAS, built-in index-based binary operators such as `GxB_FIRSTI_INT32` may be used, however. The equivalent index-unary operators are used in their place.

12.12.5 GrB_Vector_apply_IndexOp: apply an index-unary operator to a vector

```
GrB_Info GrB_apply                  // w<mask> = accum (w, op(u,y))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_IndexUnaryOp op,      // operator to apply to the entries
    const GrB_Vector u,             // first input:  vector u
    const <type> y,                 // second input: scalar y
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_apply_IndexOp_<type>` applies an index-unary operator $z = f(x, i, 0, y)$ to a vector. The scalar `y` can be a non-opaque C scalar corresponding to a built-in type, a `void *` for user-defined types, or a `GrB_Scalar`. It is otherwise identical to `GrB_Vector_apply`.

12.12.6 GrB_Matrix_apply_BinaryOp1st: apply a binary operator to a matrix; 1st scalar binding

```
GrB_Info GrB_apply                  // C<M>=accum(C,op(x,A))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_BinaryOp op,          // operator to apply to the entries
    <type> x,                       // first input:  scalar x
    const GrB_Matrix A,             // second input: matrix A
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

`GrB_Matrix_apply_BinaryOp1st_<type>` applies a binary operator $z = f(x, y)$ to a matrix, where a scalar x is bound to the first input of the operator. The scalar `x` can be a non-opaque C scalar corresponding to a built-in type, a `void *` for user-defined types, or a `GrB_Scalar`. It is otherwise identical to `GrB_Matrix_apply`.

The `op` can be any binary operator except that it cannot be a user-defined `GrB_BinaryOp` created by `GxB_BinaryOp_new_IndexOp`. For backward compatibility with prior versions of SuiteSparse:GraphBLAS, built-in index-based binary operators such as `GxB_FIRSTI_INT32` may be used, however. The equivalent index-unary operators are used in their place.

12.12.7 GrB_Matrix_apply_BinaryOp2nd: apply a binary operator to a matrix; 2nd scalar binding

```
GrB_Info GrB_apply                  // C<M>=accum(C,op(A,y))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_BinaryOp op,          // operator to apply to the entries
    const GrB_Matrix A,             // first input:  matrix A
    <type> y,                       // second input: scalar y
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

`GrB_Matrix_apply_BinaryOp2nd_<type>` applies a binary operator $z = f(x, y)$ to a matrix, where a scalar x is bound to the second input of the operator. The scalar y can be a non-opaque C scalar corresponding to a built-in type, a void * for user-defined types, or a GrB_Scalar. It is otherwise identical to GrB_Matrix_apply.

The op can be any binary operator except that it cannot be a user-defined GrB_BinaryOp created by GxB_BinaryOp_new_IndexOp. For backward compatibility with prior versions of SuiteSparse:GraphBLAS, built-in index-based binary operators such as GxB_FIRSTI_INT32 may be used, however. The equivalent index-unary operators are used in their place.

12.12.8 GrB_Matrix_apply_IndexOp: apply an index-unary operator to a matrix

```
GrB_Info GrB_apply                  // C<M>=accum(C,op(A,y))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_IndexUnaryOp op,      // operator to apply to the entries
    const GrB_Matrix A,             // first input:  matrix A
    const <type> y,                 // second input: scalar y
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

GrB_Matrix_apply_IndexOp_<type> applies an index-unary operator $z = f(x, i, j, y)$ to a matrix. The scalar y can be a non-opaque C scalar corresponding to a built-in type, a void * for user-defined types, or a GrB_Scalar. It is otherwise identical to GrB_Matrix_apply.

12.13 GrB_select: select entries based on an index-unary operator

The `GrB_select` function is the generic name for 30 specific functions, depending on whether it operates on a matrix or vector, and depending on the type of the scalar `y`: (matrix or vector) x (13 built-in types, `void *` for user-defined types, and a `GrB_Scalar`). The generic name appears in the function prototypes, but the specific function name is used when describing each variation. When discussing features that apply to both versions, the simple name `GrB_select` is used.

12.13.1 GrB_Vector_select: select entries from a vector

```
GrB_Info GrB_select                 // w<mask> = accum (w, op(u))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_IndexUnaryOp op,      // operator to apply to the entries
    const GrB_Vector u,             // first input:  vector u
    const <type> y,                 // second input: scalar y
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_Vector_select_*` applies a `GrB_IndexUnaryOp` operator to the entries of a vector. If the operator evaluates as `true` for the entry `u(i)`, it is copied to the vector `t`, or not copied if the operator evaluates to `false`. The vector `t` is then written to the result `w` via the mask/accumulator step. This operation operates on vectors just as if they were `m`-by-1 matrices, except that GraphBLAS never transposes a vector via the descriptor. Refer to the next section (12.13.2) on `GrB_Matrix_select` for more details.

12.13.2 GrB_Matrix_select: apply a select operator to a matrix

```
GrB_Info GrB_select                 // C<M>=accum(C,op(A))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_IndexUnaryOp op,      // operator to apply to the entries
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Scalar y,             // second input: scalar y
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

GrB_Matrix_select_* applies a GrB_IndexUnaryOp operator to the entries of a matrix. If the operator evaluates as true for the entry A(i,j), it is copied to the matrix T, or not copied if the operator evaluates to false. The input matrix A may be transposed first. The entries in A are typecasted into the xtype of the select operator. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

The matrix T has the same size and type as A (or the transpose of A if the input is transposed via the descriptor). The entries of T are a subset of those of A. Each entry A(i,j) of A is passed to the op, as $z = f(a_{ij}, i, j, y)$. If A is transposed first then the operator is applied to entries in the transposed matrix, A'. If z is returned as true, then the entry is copied into T, unchanged. If it returns false, the entry does not appear in T.

The action of GrB_select with the built-in index-unary operators is described in the table below. The MATLAB analogs are precise for tril and triu, but shorthand for the other operations. The MATLAB diag function returns a column with the diagonal, if A is a matrix, whereas the matrix T in GrB_select always has the same size as A (or its transpose if the GrB_INP0 is set to GrB_TRAN). In the MATLAB analog column, diag is as if it operates like GrB_select, where T is a matrix.

The following operators may be used on matrices with a user-defined type: GrB_ROWINDEX_*, GrB_COLINDEX_*, GrB_DIAGINDEX_*, GrB_TRIL, GrB_TRIU, GrB_DIAG, GrB_OFFIAG, GrB_COLLE, GrB_COLGT, GrB_ROWLE, and GrB_ROWGT.

For floating-point values, comparisons with NaN always return false. The GrB_VALUE* operators should not be used with a scalar y that is equal to NaN. For this case, create a user-defined index-unary operator that performs the test with the ANSI C isnan function instead.

GraphBLAS name	MATLAB/Octave analog	description
GrB_ROWINDEX_*	z=i+y	select A(i,j) if i != -y
GrB_COLINDEX_*	z=j+y	select A(i,j) if j != -y
GrB_DIAGINDEX_*	z=j-(i+y)	select A(i,j) if j != i+y
GrB_TRIL	z=(j<=(i+y))	select entries on or below the yth diagonal
GrB_TRIU	z=(j>=(i+y))	select entries on or above the yth diagonal
GrB_DIAG	z=(j==(i+y))	select entries on the yth diagonal
GrB_OFFDIAG	z=(j!=(i+y))	select entries not on the yth diagonal
GrB_COLLE	z=(j<=y)	select entries in columns 0 to y
GrB_COLGT	z=(j>y)	select entries in columns y+1 and above
GrB_ROWLE	z=(i<=y)	select entries in rows 0 to y
GrB_ROWGT	z=(i>y)	select entries in rows y+1 and above
GrB_VALUENE_T	z=(aij!=y)	select A(i,j) if it is not equal to y
GrB_VALUEEQ_T	z=(aij==y)	select A(i,j) is it equal to y
GrB_VALUEGT_T	z=(aij>y)	select A(i,j) is it greater than y
GrB_VALUEGE_T	z=(aij>=y)	select A(i,j) is it greater than or equal to y
GrB_VALUELT_T	z=(aij<y)	select A(i,j) is it less than y
GrB_VALUELE_T	z=(aij<=y)	select A(i,j) is it less than or equal to y

12.14 GrB_reduce: reduce to a vector or scalar

The generic function name `GrB_reduce` may be used for all specific functions discussed in this section. When the details of a specific function are discussed, the specific name is used for clarity.

12.14.1 GrB_Matrix_reduce_Monoid reduce a matrix to a vector

```
GrB_Info GrB_reduce                 // w<mask> = accum (w,reduce(A))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_Monoid monoid,        // reduce monoid for t=reduce(A)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Descriptor desc       // descriptor for w, mask, and A
) ;
```

`GrB_Matrix_reduce_Monoid` reduces a matrix to a column vector using a monoid, roughly analogous to `t = sum (A')` in MATLAB, in the default case, where `t` is a column vector. By default, the method reduces across the rows to obtain a column vector; use `GrB_TRAN` to reduce down the columns.

The input matrix `A` may be transposed first. Its entries are then typecast into the type of the `reduce` operator or monoid. The reduction is applied to all entries in `A (i,:)` to produce the scalar `t (i)`. This is done without the use of the identity value of the monoid. If the `i`th row `A (i,:)` has no entries, then `(i)` is not an entry in `t` and its value is implicit. If `A (i,:)` has a single entry, then that is the result `t (i)` and `reduce` is not applied at all for the `i`th row. Otherwise, multiple entries in row `A (i,:)` are reduced via the `reduce` operator or monoid to obtain a single scalar, the result `t (i)`.

The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

12.14.2 GrB_Vector_reduce_<type>: reduce a vector to a scalar

```
GrB_Info GrB_reduce                     // c = accum (c, reduce_to_scalar (u))
(
    <type> *c,                          // result scalar
    const GrB_BinaryOp accum,           // optional accum for c=accum(c,t)
    const GrB_Monoid monoid,            // monoid to do the reduction
    const GrB_Vector u,                 // vector to reduce
    const GrB_Descriptor desc           // descriptor (currently unused)
) ;

GrB_Info GrB_reduce                     // c = accum (c, reduce_to_scalar (u))
(
    GrB_Scalar c,                       // result scalar
    const GrB_BinaryOp accum,           // optional accum for c=accum(c,t)
    const GrB_Monoid monoid,            // monoid to do the reduction
    const GrB_Vector u,                 // vector to reduce
    const GrB_Descriptor desc           // descriptor (currently unused)
) ;
```

`GrB_Vector_reduce_<type>` reduces a vector to a scalar, analogous to `t = sum (u)` in MATLAB, except that in GraphBLAS any commutative and associative monoid can be used in the reduction.

The scalar `c` can be a pointer C type: `bool`, `int8_t`, ... `float`, `double`, or `void *` for a user-defined type, or a `GrB_Scalar`. If `c` is a `void *` pointer to a user-defined type, the type must be identical to the type of the vector `u`. This cannot be checked by GraphBLAS and thus results are undefined if the types are not the same.

If the vector `u` has no entries, that identity value of the `monoid` is copied into the scalar `t` (unless `c` is a `GrB_Scalar`, in which case `t` is an empty `GrB_Scalar`, with no entry). Otherwise, all of the entries in the vector are reduced to a single scalar using the `monoid`.

The descriptor is unused, but it appears in case it is needed in future versions of the GraphBLAS API. This function has no mask so its accumulator/mask step differs from the other GraphBLAS operations. It does not use the methods described in Section 2.3, but uses the following method instead.

If `accum` is `NULL`, then the scalar `t` is typecast into the type of `c`, and `c = t` is the final result. Otherwise, the scalar `t` is typecast into the `ytype` of the `accum` operator, and the value of `c` (on input) is typecast into the `xtype` of the `accum` operator. Next, the scalar `z = accum (c,t)` is computed, of the `ztype` of the `accum` operator. Finally, `z` is typecast into the final result, `c`.

If `c` is a non-opaque scalar, no error message can be returned by `GrB_error`. If `c` is a `GrB_Scalar`, then `GrB_error(&err,c)` can be used to return an error string, if an error occurs.

12.14.3 GrB_Matrix_reduce_<type>: reduce a matrix to a scalar

```
GrB_Info GrB_reduce                 // c = accum (c, reduce_to_scalar (A))
(
    <type> *c,                      // result scalar
    const GrB_BinaryOp accum,       // optional accum for c=accum(c,t)
    const GrB_Monoid monoid,        // monoid to do the reduction
    const GrB_Matrix A,             // matrix to reduce
    const GrB_Descriptor desc       // descriptor (currently unused)
) ;

GrB_Info GrB_reduce                 // c = accum (c, reduce_to_scalar (A))
(
    GrB_Scalar c,                   // result scalar
    const GrB_BinaryOp accum,       // optional accum for c=accum(c,t)
    const GrB_Monoid monoid,        // monoid to do the reduction
    const GrB_Matrix A,             // matrix to reduce
    const GrB_Descriptor desc       // descriptor (currently unused)
) ;
```

`GrB_Matrix_reduce_<type>` reduces a matrix `A` to a scalar, roughly analogous to `t = sum (A (:))` in MATLAB. This function is identical to reducing a vector to a scalar, since the positions of the entries in a matrix or vector have no effect on the result. Refer to the reduction to scalar described in the previous Section 12.14.2.

12.15 GrB_transpose: transpose a matrix

```
GrB_Info GrB_transpose              // C<Mask> = accum (C, A')
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Descriptor desc       // descriptor for C, Mask, and A
) ;
```

`GrB_transpose` transposes a matrix `A`, just like the array transpose `T = A.'` in MATLAB. The internal result matrix `T = A'` (or merely `T = A` if `A` is transposed via the descriptor) has the same type as `A`. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3, which typecasts `T` as needed and applies the mask and accumulator.

To be consistent with the rest of the GraphBLAS API regarding the descriptor, the input matrix `A` may be transposed first by setting the `GrB_INP0` setting to `GrB_TRAN`. This results in a double transpose, and thus `A` is not transposed is computed.

12.16 GrB_kronecker: Kronecker product

```
GrB_Info GrB_kronecker              // C<Mask> = accum (C, kron(A,B))
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const <operator> op,            // defines '*' for T=kron(A,B)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Matrix B,             // second input: matrix B
    const GrB_Descriptor desc       // descriptor for C, Mask, A, and B
) ;
```

`GrB_kronecker` computes the Kronecker product, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathrm{kron}(\mathbf{A},\mathbf{B})$ where

$$\mathrm{kron}(\mathbf{A},\mathbf{B}) = \left[\begin{array}{ccc} a_{00} \otimes \mathbf{B} & \ldots & a_{0,n-1} \otimes \mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} \otimes \mathbf{B} & \ldots & a_{m-1,n-1} \otimes \mathbf{B} \end{array} \right]$$

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. The `op` may be a binary operator created by `GxB_BinaryOp_new_IndexOp`.

13 Printing GraphBLAS objects

The ten different objects handled by SuiteSparse:GraphBLAS are all opaque, although nearly all of their contents can be extracted via methods such as `GrB_Matrix_extractTuples`, `GrB_Matrix_extractElement`, `GrB_get`, and so on. The GraphBLAS C API has no mechanism for printing all the contents of GraphBLAS objects, but this is helpful for debugging. Ten type-specific methods and two type-generic methods are provided:

`GxB_Type_fprint`	print and check a `GrB_Type`
`GxB_UnaryOp_fprint`	print and check a `GrB_UnaryOp`
`GxB_BinaryOp_fprint`	print and check a `GrB_BinaryOp`
`GxB_IndexUnaryOp_fprint`	print and check a `GrB_IndexUnaryOp`
`GxB_IndexBinaryOp_fprint`	print and check a `GxB_IndexBinaryOp`
`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_Context_fprint`	print and check a `GxB_Context`
`GxB_Matrix_fprint`	print and check a `GrB_Matrix`
`GxB_Vector_fprint`	print and check a `GrB_Vector`
`GxB_Scalar_fprint`	print and check a `GrB_Scalar`
`GxB_fprint`	print/check any object to a file
`GxB_print`	print/check any object to `stdout`

These methods do not modify the status of any object, and thus they cannot return an error string for use by `GrB_error`.

If a matrix or vector has not been completed, the pending computations are guaranteed to *not* be performed. The reason is simple. It is possible for a bug in the user application (such as accessing memory outside the bounds of an array) to mangle the internal content of a GraphBLAS object, and the `GxB_*print` methods can be helpful tools to track down this bug. If `GxB_*print` attempted to complete any computations prior to printing or checking the contents of the matrix or vector, then further errors could occur, including a segfault.

By contrast, GraphBLAS methods and operations that return values into user-provided arrays or variables might finish pending operations before the return these values, and this would change their state. Since they do not change the state of any object, the `GxB_*print` methods provide a useful method for debugging, and for a quick understanding of what GraphBLAS is computing while developing a user application.

Each of the methods has a parameter of type GxB_Print_Level that specifies the amount to print:

```
typedef enum
{
    GxB_SILENT = 0,      // nothing is printed, just check the object
    GxB_SUMMARY = 1,     // print a terse summary
    GxB_SHORT = 2,       // short description, about 30 entries of a matrix
    GxB_COMPLETE = 3,    // print the entire contents of the object
    GxB_SHORT_VERBOSE = 4,    // GxB_SHORT but with "%.15g" for doubles
    GxB_COMPLETE_VERBOSE = 5  // GxB_COMPLETE but with "%.15g" for doubles
}
GxB_Print_Level ;
```

The ten type-specific functions include an additional argument, the name string. The name is printed at the beginning of the display (assuming the print level is not GxB_SILENT) so that the object can be more easily identified in the output. For the type-generic methods GxB_fprint and GxB_print, the name string is the variable name of the object itself.

If the file f is NULL, stdout is used. If name is NULL, it is treated as the empty string. These are not error conditions.

The methods check their input objects carefully and extensively, even when pr is equal to GxB_SILENT. The following error codes can be returned:

- GrB_SUCCESS: object is valid
- GrB_UNINITIALIZED_OBJECT: object is not initialized
- GrB_INVALID_OBJECT: object is not valid
- GrB_NULL_POINTER: object is a NULL pointer
- GrB_INVALID_VALUE: fprintf returned an I/O error.

The content of any GraphBLAS object is opaque, and subject to change. As a result, the exact content and format of what is printed is implementation-dependent, and will change from version to version of SuiteSparse:GraphBLAS. Do not attempt to rely on the exact content or format by trying to parse the resulting output via another program. The intent of these functions is to produce a report of an object for visual inspection. If the user application needs to extract content from a GraphBLAS matrix or vector, use GrB_*_extractTuples or the import/export methods instead.

GraphBLAS matrices and vectors are zero-based, where indices of an n-by-n matrix are in the range 0 to $n-1$. However, MATLAB, Octave, and Julia prefer to print their matrices and vectors as one-based. To enable 1-based printing, use `GrB_set (GrB_GLOBAL, true, GxB_PRINT_1BASED)`. Printing is done as zero-based by default.

13.1 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
    int pr,                         // print level (GxB_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)`.

13.2 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
    int pr,                         // print level (GxB_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)`.

13.3 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
    int pr,                         // print level (GxB_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`.

13.4 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
    int pr,                         // print level (GxB_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`.

13.5 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
    int pr,                         // print level (GxB_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`.

13.6 GxB_IndexUnaryOp_fprint: Print a GrB_IndexUnaryOp

```
GrB_Info GxB_IndexUnaryOp_fprint    // print and check a GrB_IndexUnaryOp
(
    GrB_IndexUnaryOp op,            // object to print and check
    const char *name,               // name of the object
    int pr,                         // print level (GxB_Print_Level)
    FILE *f                         // file for output
) ;
```

For example, `GrB_IndexUnaryOp_fprint (GrB_TRIL, "tril", GxB_COMPLETE, f)` prints the `GrB_TRIL` index-unary operator to the file `f`.

13.7 GxB_IndexBinaryOp_fprint: Print a GxB_IndexBinaryOp

```
GrB_Info GxB_IndexBinaryOp_fprint   // print and check a GxB_IndexBinaryOp
(
    GxB_IndexBinaryOp op,           // object to print and check
    const char *name,               // name of the object
    int pr,                         // print level (GxB_Print_Level)
    FILE *f                         // file for output
) ;
```

13.8 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
    int pr,                         // print level (GxB_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`.

13.9 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
    int pr,                         // print level (GxB_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`.

13.10 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
    int pr,                         // print level (GxB_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`.

13.11 GxB_Context_fprint: Print a GxB_Context

```
GrB_Info GxB_Context_fprint         // print and check a GxB_Context
(
    GxB_Context Context,            // object to print and check
    const char *name,               // name of the object
    int pr,                         // print level (GxB_Print_Level)
    FILE *f                         // file for output
) ;
```

This method can be used to print the context created for a user thread, or the contents of the `GxB_CONTEXT_WORLD` object.

13.12 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
    int pr,                         // print level (GxB_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`.

13.13 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
    int pr,                         // print level (GxB_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`.

13.14 GxB_Scalar_fprint: Print a GrB_Scalar

```
GrB_Info GxB_Scalar_fprint          // print and check a GrB_Scalar
(
    GrB_Scalar s,                   // object to print and check
    const char *name,               // name of the object
    int pr,                         // print level (GxB_Print_Level)
    FILE *f                         // file for output
) ;
```

For example, `GxB_Scalar_fprint (s, "my scalar", GxB_SHORT, f)` prints a short description of the scalar `s` to the file `f`.

13.15 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.

14 Matrix and Vector iterators

The `GxB_Iterator` is an object that allows user applications to iterate over the entries of a matrix or vector, one entry at a time. Iteration can be done in a linear manner (analogous to reading a file one entry at a time, from start to finish), or in a random-access pattern (analogous to the `fseek` method for repositioning the access to file to a different position).

Multiple iterators can be used on a single matrix or vector, even in parallel by multiple user threads. While a matrix or vector is being used with an iterator, the matrix or vector must not be modified. Doing so will lead to undefined results.

Since accessing a matrix or vector via an iterator requires many calls to the iterator methods, they must be very fast. Error checking is skipped, except for the methods that create, attach, or free an iterator. Methods that advance an iterator or that access values or indices from a matrix or vector do not return error conditions. Instead, they have well-defined preconditions that must be met (and which should be checked by the user application). If those preconditions are not met, results are undefined.

The iterator methods are implemented in SuiteSparse:GraphBLAS as both macros (via `#define`) and as functions of the same name that appear in the compiled `libgraphblas.so` library. This requires that the opaque contents of the iterator object be defined in `GraphBLAS.h` itself. The user application must not access these contents directly, but can only do so safely via the iterator methods provided by SuiteSparse:GraphBLAS.

The iterator object can be used in one of four sets of methods, for four different access patterns:

1. *row iterator*: iterates across the rows of a matrix, and then within each row to access the entries in a given row. Accessing all the entries of a matrix using a row iterator requires an outer loop (for the rows) and an inner loop (for the entries in each row). A matrix can be accessed via a row iterator only if its format (determined by `GrB_get (A, &fmt, GrB_STORAGE_ORIENTATION_HINT)`) is by-row (that is, `GrB_ROWMAJOR`). See Section 10.

2. *column iterator*: iterates across the columns of a matrix, and then within each column to access the entries in a given column. Accessing all the entries of a matrix using a column iterator requires an outer loop

(for the columns) and an inner loop (for the entries in each column). A matrix can be accessed via a column iterator only if its format (determined by `GrB_get (A, &fmt, GrB_STORAGE_ORIENTATION_HINT)`) is by-column (that is, `GrB_COLMAJOR`). See Section 10.

3. *entry iterator*: iterates across the entries of a matrix. Accessing all the entries of a matrix using an entry iterator requires just a single loop. Any matrix can be accessed with an entry iterator.

4. *vector iterator*: iterates across the entries of a vector. Accessing all the entries of a vector using a vector iterator requires just a single loop. Any vector can be accessed with a vector iterator.

14.1 Creating and destroying an iterator

The process for using an iterator starts with the creation of an iterator, with `GxB_Iterator_new`. This method creates an `iterator` object but does not *attach* it to any specific matrix or vector:

```
GxB_Iterator iterator ;
GxB_Iterator_new (&iterator) ;
```

When finished, the `iterator` is freed with either of these methods:

```
GrB_free (&iterator) ;
GxB_Iterator_free (&iterator) ;
```

14.2 Attaching an iterator to a matrix or vector

This new `iterator` object can be *attached* to any matrix or vector, and used as a row, column, or entry iterator for any matrix, or as an iterator for any vector. The `iterator` can be used in any of these methods before it is freed, but with just one access method at a time.

Once it is created, the `iterator` must be attached to a matrix or vector. This process also selects the method by which the `iterator` will be used for a matrix. Each of the four `GxB_*Iterator_attach` methods returns a `GrB_Info` result.

1. *row iterator*:

```
GrB_Info info = GxB_rowIterator_attach (iterator, A, desc) ;
```

2. *column iterator*:

```
GrB_Info info = GxB_colIterator_attach (iterator, A, desc) ;
```

3. *entry iterator*:

```
GrB_Info info = GxB_Matrix_Iterator_attach (iterator, A, desc) ;
```

4. *vector iterator*:

```
GrB_Info info = GxB_Vector_Iterator_attach (iterator, v, desc) ;
```

On input to `GxB_*Iterator_attach`, the `iterator` must already exist, having been created by `GxB_Iterator_new`. If the `iterator` is already attached to a matrix or vector, it is detached and then attached to the given matrix `A` or vector `v`.

The return values for row/column methods are:

- `GrB_SUCCESS`: if the `iterator` is successfully attached to the matrix `A`.
- `GrB_NULL_POINTER`: if the `iterator` or `A` are NULL.
- `GrB_INVALID_OBJECT`: if the matrix `A` is invalid.
- `GrB_NOT_IMPLEMENTED`: if the matrix `A` cannot be iterated in the requested access method (row iterators require the matrix to be held by-row, and column iterators require the matrix to be held by-column).
- `GrB_OUT_OF_MEMORY`: if the method runs out of memory.

The other two methods (entry iterator for matrices, or the vector iterator) return the same error codes, except that they do not return `GrB_NOT_IMPLEMENTED`.

14.3 Seeking to an arbitrary position

Attaching the `iterator` to a matrix or vector does not define a specific position for the `iterator`. To use the `iterator`, a single call to the corresponding *seek* method is required. These `GxB*_Iterator_*seek*` methods may also be used later on to change the position of the iterator arbitrarily.

1. *row iterator*:

```
GrB_Info info = GxB_rowIterator_seekRow (iterator, row) ;
GrB_Index kount = GxB_rowIterator_kount (iterator) ;
GrB_Info info = GxB_rowIterator_kseek (iterator, k) ;
```

These methods move a row iterator to a specific row, defined in one of two ways: (1) the row index itself (in range 0 to `nrows`-1), or (2) by specifying `k`, which moves the iterator to the `k`th *explicit* row (in the range 0 to `kount`-1). For sparse, bitmap, or full matrices, these two methods are identical. For hypersparse matrices, not all rows are present in the data structure; these *implicit* rows are skipped and not included in the `kount`. Implicit rows contain no entries. The `GxB_rowIterator_kount` method returns the `kount` of the matrix, where `kount` is equal to `nrows` for sparse, bitmap, and matrices, and `kount` $\leq$ `nrows` for hypersparse matrices. All three methods listed above can be used for any row iterator.

The `GxB_rowIterator_*seek*` methods return `GrB_SUCCESS` if the iterator has been moved to a row that contains at least one entry, `GrB_NO_VALUE` if the row has no entries, or `GxB_EXHAUSTED` if the row is out of bounds (`row` $\geq$ `nrows` or if `k` $\geq$ `kount`). None of these return conditions are errors; they are all informational.

For sparse, bitmap, and full matrices, `GxB_rowIterator_seekRow` always moves to the given row. For hypersparse matrices, if the requested row is implicit, the iterator is moved to the first explicit row following it. If no such row exists, the iterator is exhausted and `GxB_EXHAUSTED` is returned. The `GxB_rowIterator_kseek` method always moves to the `k`th explicit row, for any matrix. Use `GxB_rowIterator_getRowIndex`, described below, to determine the row index of the current position.

Precondition: on input, the `iterator` must have been successfully attached to a matrix via a prior call to `GxB_rowIterator_attach`. Results are undefined if this precondition is not met.

2. *column iterator*:

```
GrB_Info info = GxB_colIterator_seekCol (iterator, col) ;
GrB_Index kount = GxB_colIterator_kount (iterator) ;
GrB_Info info = GxB_colIterator_kseek (iterator, k) ;
```

These methods move a column iterator to a specific column, defined in one of two ways: (1) the column index itself (in range 0 to `ncols`-1), or

(2) by specifying `k`, which moves the iterator to the `k`th *explicit* column (in the range 0 to `kount`-1). For sparse, bitmap, or full matrices, these two methods are identical. For hypersparse matrices, not all columns are present in the data structure; these *implicit* columns are skipped and not included in the `kount`. Implicit columns contain no entries. The `GxB_colIterator_kount` method returns the `kount` of the matrix, where `kount` is equal to `ncols` for sparse, bitmap, and matrices, and `kount` $\leq$ `ncols` for hypersparse matrices. All three methods listed above can be used for any column iterator.

The `GxB_colIterator_*seek*` methods return `GrB_SUCCESS` if the iterator has been moved to a column that contains at least one entry, `GrB_NO_VALUE` if the column has no entries, or `GxB_EXHAUSTED` if the column is out of bounds (`col` $\geq$ `ncols` or `k` $\geq$ `kount`). None of these return conditions are errors; they are all informational.

For sparse, bitmap, and full matrices, `GxB_colIterator_seekCol` always moves to the given column. For hypersparse matrices, if the requested column is implicit, the iterator is moved to the first explicit column following it. If no such column exists, the iterator is exhausted and `GxB_EXHAUSTED` is returned. The `GxB_colIterator_kseek` method always moves to the `k`th explicit column, for any matrix. Use `GxB_colIterator_getColIndex`, described below, to determine the column index of the current position.

Precondition: on input, the `iterator` must have been successfully attached to a matrix via a prior call to `GxB_colIterator_attach`. Results are undefined if this precondition is not met.

3. *entry iterator*:

```
GrB_Info info = GxB_Matrix_Iterator_seek (iterator, p) ;
GrB_Index pmax = GxB_Matrix_Iterator_getpmax (iterator) ;
GrB_Index p = GxB_Matrix_Iterator_getp (iterator);
```

The `GxB_Matrix_Iterator_seek` method moves the `iterator` to the given position `p`, which is in the range 0 to `pmax`-1, where the value of `pmax` is obtained from `GxB_Matrix_Iterator_getpmax`. For sparse, hypersparse, and full matrices, `pmax` is the same as `nvals` returned by `GrB_Matrix_nvals`. For bitmap matrices, `pmax` is equal to `nrows*ncols`.

If p $\geq$ pmax, the iterator is exhausted and GxB_EXHAUSTED is returned. Otherwise, GrB_SUCCESS is returned.

All entries in the matrix are given an ordinal position, p. Seeking to position p will either move the iterator to that particular position, or to the next higher position containing an entry if there is entry at position p. The latter case only occurs for bitmap matrices. Use GxB_Matrix_Iterator_getp to determine the current position of the iterator.

Precondition: on input, the iterator must have been successfully attached to a matrix via a prior call to GxB_Matrix_Iterator_attach. Results are undefined if this precondition is not met.

4. *vector iterator*:

```
GrB_Info info = GxB_Vector_Iterator_seek (iterator, p) ;
GrB_Index pmax = GxB_Vector_Iterator_getpmax (iterator) ;
GrB_Index p = GxB_Vector_Iterator_getp (iterator);
```

The GxB_Vector_Iterator_seek method is identical to the entry iterator of a matrix, but applied to a GrB_Vector instead.

Precondition: on input, the iterator must have been successfully attached to a vector via a prior call to GxB_Vector_Iterator_attach. Results are undefined if this precondition is not met.

14.4 Advancing to the next position

For best performance, the *seek* methods described above should be used with care, since some of them require $O(\log n)$ time. The fastest method for changing the position of the iterator is the corresponding *next* method, described below for each iterator:

1. *row iterator*: To move to the next row.

```
GrB_Info info = GxB_rowIterator_nextRow (iterator) ;
```

The row iterator is a 2-dimensional iterator, requiring an outer loop and an inner loop. The outer loop iterates over the rows of the matrix, using GxB_rowIterator_nextRow to move to the next row. If the matrix is hypersparse, the next row is always an explicit row; implicit rows are skipped. The return conditions are identical to GxB_rowIterator_seekRow.

Preconditions: on input, the row iterator must already be attached to a matrix via a prior call to GxB_rowIterator_attach, and the iterator must be at a specific row, via a prior call to GxB_rowIterator_*seek* or GxB_rowIterator_nextRow. Results are undefined if these conditions are not met.

2. *row iterator*: To move to the next entry within a row.

```
GrB_Info info = GxB_rowIterator_nextCol (iterator) ;
```

The row iterator is moved to the next entry in the current row. The method returns GrB_NO_VALUE if the end of the row is reached. The iterator does not move to the next row in this case. The method returns GrB_SUCCESS if the iterator has been moved to a specific entry in the current row.

Preconditions: the same as GxB_rowIterator_nextRow.

3. *column iterator*: To move to the next column

```
GrB_Info info = GxB_colIterator_nextCol (iterator) ;
```

The column iterator is a 2-dimensional iterator, requiring an outer loop and an inner loop. The outer loop iterates over the columns of the matrix, using GxB_colIterator_nextCol to move to the next column. If the matrix is hypersparse, the next column is always an explicit column; implicit columns are skipped. The return conditions are identical to GxB_colIterator_seekCol.

Preconditions: on input, the column iterator must already be attached to a matrix via a prior call to GxB_colIterator_attach, and the iterator must be at a specific column, via a prior call to GxB_colIterator_*seek* or GxB_colIterator_nextCol. Results are undefined if these conditions are not met.

4. *column iterator*: To move to the next entry within a column.

```
GrB_Info info = GxB_colIterator_nextRow (iterator) ;
```

The column iterator is moved to the next entry in the current column. The method returns GrB_NO_VALUE if the end of the column is reached. The iterator does not move to the next column in this case. The method returns GrB_SUCCESS if the iterator has been moved to a specific entry in the current column.

Preconditions: the same as GxB_colIterator_nextCol.

5. *entry iterator*: To move to the next entry.

```
GrB_Info info = GxB_Matrix_Iterator_next (iterator) ;
```

This method moves an iterator to the next entry of a matrix. It returns GrB_SUCCESS if the iterator is at an entry that exists in the matrix, or GrB_EXHAUSTED otherwise.

Preconditions: on input, the entry iterator must be already attached to a matrix via GxB_Matrix_Iterator_attach, and the position of the iterator must also have been defined by a prior call to GxB_Matrix_Iterator_seek or GxB_Matrix_Iterator_next. Results are undefined if these conditions are not met.

6. *vector iterator*: To move to the next entry.

```
GrB_Info info = GxB_Vector_Iterator_next (iterator) ;
```

This method moves an iterator to the next entry of a vector. It returns GrB_SUCCESS if the iterator is at an entry that exists in the vector, or GrB_EXHAUSTED otherwise.

Preconditions: on input, the iterator must be already attached to a vector via GxB_Vector_Iterator_attach, and the position of the iterator must also have been defined by a prior call to GxB_Vector_Iterator_seek or GxB_Vector_Iterator_next. Results are undefined if these conditions are not met.

14.5 Accessing the indices of the current entry

Once the iterator is attached to a matrix or vector, and is placed in position at an entry in the matrix or vector, the indices and value of this entry can be obtained. The methods for accessing the value of the entry are described in Section 14.6. Accessing the indices is performed with four different sets of methods, depending on which access pattern is in use, described below:

1. *row iterator*: To get the current row index.

```
GrB_Index i = GxB_rowIterator_getRowIndex (iterator) ;
```

   The method returns `nrows(A)` if the iterator is exhausted, or the current row index `i` otherwise. There need not be any entry in the current row. Zero is returned if the iterator is attached to the matrix but `GxB_rowIterator_*seek*` has not been called, but this does not mean the iterator is positioned at row zero.

   Preconditions: on input, the iterator must be already successfully attached to matrix as a row iterator via `GxB_rowIterator_attach`. Results are undefined if this condition is not met.

2. *row iterator*: To get the current column index.

```
GrB_Index j = GxB_rowIterator_getColIndex (iterator) ;
```

   Preconditions: on input, the iterator must be already successfully attached to matrix as a row iterator via `GxB_rowIterator_attach`, and in addition, the row iterator must be positioned at a valid entry present in the matrix. That is, the last call to `GxB_rowIterator_*seek*` or `GxB_rowIterator_*next*`, must have returned `GrB_SUCCESS`. Results are undefined if these conditions are not met.

3. *column iterator*: To get the current column index.

```
GrB_Index j = GxB_colIterator_getColIndex (iterator) ;
```

   The method returns `ncols(A)` if the iterator is exhausted, or the current column index `j` otherwise. There need not be any entry in the current column. Zero is returned if the iterator is attached to the matrix but `GxB_colIterator_*seek*` has not been called, but this does not mean the iterator is positioned at column zero.

   Precondition: on input, the iterator must be already successfully attached to matrix as a column iterator via `GxB_colIterator_attach`. Results are undefined if this condition is not met.

4. *column iterator*: To get the current row index.

```
GrB_Index i = GxB_colIterator_getRowIndex (iterator) ;
```

Preconditions: on input, the iterator must be already successfully attached to matrix as a column iterator via GxB_colIterator_attach, and in addition, the column iterator must be positioned at a valid entry present in the matrix. That is, the last call to GxB_colIterator_*seek* or GxB_colIterator_*next*, must have returned GrB_SUCCESS. Results are undefined if these conditions are not met.

5. *entry iterator*: To get the current row and column index.

```
GrB_Index i, j ;
GxB_Matrix_Iterator_getIndex (iterator, &i, &j) ;
```

   Returns the row and column index of the current entry.

   Preconditions: on input, the entry iterator must be already attached to a matrix via GxB_Matrix_Iterator_attach, and the position of the iterator must also have been defined by a prior call to GxB_Matrix_Iterator_seek or GxB_Matrix_Iterator_next, with a return value of GrB_SUCCESS. Results are undefined if these conditions are not met.

6. *vector iterator*: To get the current index.

```
GrB_Index i = GxB_Vector_Iterator_getIndex (iterator) ;
```

   Returns the index of the current entry.

   Preconditions: on input, the entry iterator must be already attached to a matrix via GxB_Vector_Iterator_attach, and the position of the iterator must also have been defined by a prior call to GxB_Vector_Iterator_seek or GxB_Vector_Iterator_next, with a return value of GrB_SUCCESS. Results are undefined if these conditions are not met.

14.6 Accessing the value of the current entry

So far, all methods that create or use an iterator have been split into four sets of methods, for the row, column, or entry iterators attached to a matrix, or for a vector iterator. Accessing the value is different. All four iterators use the same set of methods to access the value of their current entry. These methods return the value of the current entry at the position determined by the iterator. The return value can of course be typecasted using standard C syntax once the value is returned to the caller.

Preconditions: on input, the prior call to GxB_*Iterator_*seek*, or GxB_*Iterator_*next* must have returned GrB_SUCCESS, indicating that the iterator is at a valid current entry for either a matrix or vector. No typecasting is permitted, in the sense that the method name must match the type of the matrix or vector. Results are undefined if these conditions are not met.

```
// for built-in types:
bool       value = GxB_Iterator_get_BOOL (iterator) ;
int8_t     value = GxB_Iterator_get_INT8 (iterator) ;
int16_t    value = GxB_Iterator_get_INT16 (iterator) ;
int32_t    value = GxB_Iterator_get_INT32 (iterator) ;
int64_t    value = GxB_Iterator_get_INT64 (iterator) ;
uint8_t    value = GxB_Iterator_get_UINT8 (iterator) ;
uint16_t   value = GxB_Iterator_get_UINT16 (iterator) ;
uint32_t   value = GxB_Iterator_get_UINT32 (iterator) ;
uint64_t   value = GxB_Iterator_get_UINT64 (iterator) ;
float      value = GxB_Iterator_get_FP32 (iterator) ;
double     value = GxB_Iterator_get_FP64 (iterator) ;
GxB_FC32_t value = GxB_Iterator_get_FC32 (iterator) ;
GxB_FC64_t value = GxB_Iterator_get_FC64 (iterator) ;

// for user-defined types:
<type> value ;
GxB_Iterator_get_UDT (iterator, (void *) &value) ;
```

14.7 Example: row iterator for a matrix

The following example uses a row iterator to access all of the entries in a matrix A of type GrB_FP64. Note the inner and outer loops. The outer loop iterates over all rows of the matrix. The inner loop iterates over all entries in the row i. This access pattern requires the matrix to be held by-row, but otherwise it works for any matrix. If the matrix is held by-column, then use the column iterator methods instead.

```
// create an iterator
GxB_Iterator iterator ;
GxB_Iterator_new (&iterator) ;
// attach it to the matrix A, known to be type GrB_FP64
GrB_Info info = GxB_rowIterator_attach (iterator, A, NULL) ;
if (info < 0) { handle the failure ... }
// seek to A(0,:)
info = GxB_rowIterator_seekRow (iterator, 0) ;
while (info != GxB_EXHAUSTED)
{
    // iterate over entries in A(i,:)
    GrB_Index i = GxB_rowIterator_getRowIndex (iterator) ;
    while (info == GrB_SUCCESS)
    {
        // get the entry A(i,j)
        GrB_Index j = GxB_rowIterator_getColIndex (iterator) ;
        double  aij = GxB_Iterator_get_FP64 (iterator) ;
        // move to the next entry in A(i,:)
        info = GxB_rowIterator_nextCol (iterator) ;
    }
    // move to the next row, A(i+1,:), or a subsequent one if i+1 is implicit
    info = GxB_rowIterator_nextRow (iterator) ;
}
GrB_free (&iterator) ;
```

14.8 Example: column iterator for a matrix

The column iterator is analgous to the row iterator.

The following example uses a column iterator to access all of the entries in a matrix A of type GrB_FP64. The outer loop iterates over all columns of the matrix. The inner loop iterates over all entries in the column j. This access pattern requires the matrix to be held by-column, but otherwise it works for any matrix. If the matrix is held by-row, then use the row iterator methods instead.

```
// create an iterator
GxB_Iterator iterator ;
GxB_Iterator_new (&iterator) ;
// attach it to the matrix A, known to be type GrB_FP64
GrB_Info info = GxB_colIterator_attach (iterator, A, NULL) ;
// seek to A(:,0)
info = GxB_colIterator_seekCol (iterator, 0) ;
while (info != GxB_EXHAUSTED)
{
    // iterate over entries in A(:,j)
    GrB_Index j = GxB_colIterator_getColIndex (iterator) ;
    while (info == GrB_SUCCESS)
    {
        // get the entry A(i,j)
        GrB_Index i = GxB_colIterator_getRowIndex (iterator) ;
        double  aij = GxB_Iterator_get_FP64 (iterator) ;
        // move to the next entry in A(:,j)
        info = GxB_colIterator_nextRow (iterator) ;
    }
    // move to the next column, A(:,j+1), or a subsequent one if j+1 is implicit
    info = GxB_colIterator_nextCol (iterator) ;
}
GrB_free (&iterator) ;
```

14.9 Example: entry iterator for a matrix

The entry iterator allows for a simpler access pattern, with a single loop, but using a row or column iterator is faster. The method works for any matrix.

```
// create an iterator
GxB_Iterator iterator ;
GxB_Iterator_new (&iterator) ;
// attach it to the matrix A, known to be type GrB_FP64
GrB_Info info = GxB_Matrix_Iterator_attach (iterator, A, NULL) ;
if (info < 0) { handle the failure ... }
// seek to the first entry
info = GxB_Matrix_Iterator_seek (iterator, 0) ;
while (info != GxB_EXHAUSTED)
{
    // get the entry A(i,j)
    GrB_Index i, j ;
    GxB_Matrix_Iterator_getIndex (iterator, &i, &j) ;
    double aij = GxB_Iterator_get_FP64 (iterator) ;
    // move to the next entry in A
    info = GxB_Matrix_Iterator_next (iterator) ;
}
GrB_free (&iterator) ;
```

14.10 Example: vector iterator

A vector iterator is used much like an entry iterator for a matrix.

```
// create an iterator
GxB_Iterator iterator ;
GxB_Iterator_new (&iterator) ;
// attach it to the vector v, known to be type GrB_FP64
GrB_Info info = GxB_Vector_Iterator_attach (iterator, v, NULL) ;
if (info < 0) { handle the failure ... }
// seek to the first entry
info = GxB_Vector_Iterator_seek (iterator, 0) ;
while (info != GxB_EXHAUSTED)
{
    // get the entry v(i)
    GrB_Index i = GxB_Vector_Iterator_getIndex (iterator) ;
    double vi = GxB_Iterator_get_FP64 (iterator) ;
    // move to the next entry in v
    info = GxB_Vector_Iterator_next (iterator) ;
}
GrB_free (&iterator) ;
```

14.11 Performance

I have benchmarked the performance of the row and column iterators to compute `y=0` and then `y+=A*x` where `y` is a dense vector and `A` is a sparse matrix, using a single thread. The row and column iterators are very fast, sometimes only 1% slower than calling `GrB_mxv` to compute the same thing (also assuming a single thread), for large problems. For sparse matrices that average just 1 or 2 entries per row, the row iterator can be about 30% slower than `GrB_mxv`, likely because of the slightly higher complexity of moving from one row to the next using these methods.

It is possible to split up the problem for multiple user threads, each with its own iterator. Given the low overhead of the row and column iterator for a single thread, this should be very fast. Care must be taken to ensure a good load balance. Simply spliting up the rows of a matrix and giving the same number of rows to each user thread can result in imbalanced work. This is handled internally in `GrB_*` methods, but enabling parallelism when using iterators is the responsibility of the user application.

The entry iterators are easier to use but harder to implement. The methods must internally fuse both inner and outer loops so that the user application can use a single loop. As a result, the computation `y+=A*x` can be up to 4x slower (about 2x typical) than when using `GrB_mxv` with a single thread.

To obtain the best performace possible, many of the iterator methods are implemented as macros in `GraphBLAS.h`. Using macros is the default, giving typical C and C++ applications access to the fastest methods possible.

To ensure access to these methods when not using the macros, these methods are also defined as regular functions that appear in the compiled `libgraphblas.so` library with the same name as the macros. Applications that cannot use the macro versions can `#undef` the macros after the `#include <GraphBLAS.h>` statement, and then they would access the regular compiled functions in `libgraphblas.so`. This non-macro approach is not the default, and the iterator methods may be slightly slower.

15 Iso-Valued Matrices and Vectors

The GraphBLAS C API states that the entries in all `GrB_Matrix` and `GrB_Vector` objects have a numerical value, with either a built-in or user-defined type. Representing an unweighted graph requires a value to be placed on each edge, typically $a_{ij} = 1$. Adding a structure-only data type would not mix well with the rest of GraphBLAS, where all operators, monoids, and semirings need to operate on a value, of some data type. And yet unweighted graphs are very important in graph algorithms.

The solution is simple, and exploiting it in SuiteSparse:GraphBLAS requires nearly no extensions to the GraphBLAS C API. SuiteSparse:GraphBLAS can often detect when the user application is creating a matrix or vector where all entries in the sparsity pattern take on the same numerical value.

For example, $\mathbf{C}\langle\mathbf{C}\rangle = 1$, when the mask is structural, sets all entries in $\mathbf{C}$ to the value 1. SuiteSparse:GraphBLAS detects this, and performs this assignment in $O(1)$ time. It stores a single copy of this "iso-value" and sets an internal flag in the opaque data structure for $\mathbf{C}$, which states that all entries in the pattern of $\mathbf{C}$ are equal to 1. This saves both time and memory and allows for the efficient representation of sparse adjacency matrices of unweighted graphs, yet does not change the C API. To the user application, it still appears that $\mathbf{C}$ has `nvals(C)` entries, all equal to 1.

Creating and operating on iso-valued matrices (or just *iso matrices* for short) is significantly faster than creating matrices with different data values. A matrix that is iso requires only $O(1)$ space for its numerical values. The sparse and hypersparse formats require an additional $O(n+e)$ or $O(e)$ integer space to hold the pattern of an n-by-n matrix `C`, respectively, and a matrix `C` in bitmap format requires $O(n^2)$ space for the bitmap. A full matrix requires no integer storage, so a matrix that is both iso and full requires only $O(1)$ space, regardless of its dimension.

The sections below a describe the methods that can be used to create iso matrices and vectors. Let a, b, and c denote the iso values of `A`, `B`, and `C`, respectively.

15.1 Using iso matrices and vectors in a graph algorithm

There are two primary useful ways to use iso-valued matrices and vectors: (1) as iso sparse/hypersparse adjacency matrices for unweighted graphs, and

(2) as iso full matrices or vectors used with operations that do not need to access all of the content of the iso full matrix or vector.

In the first use case, simply create a GrB_Matrix with values that are all the same (those in the sparsity pattern). The GxB_Matrix_build_Scalar method can be used for this, since it guarantees that the time and work spent on the numerical part of the array is only $O(1)$. The method still must spend $O(e)$ or $O(e \log e)$ time on the integer arrays that represent the sparsity pattern, but the reduction in time and work on the numerical part of the matrix will improve performance.

The use of GxB_Matrix_build_Scalar is optional. Matrices can also be constructed with GrB* methods. In particular, GrB_Matrix_build_* can be used. It first builds a non-iso matrix and then checks if all of the values are the same, after assembling any duplicate entries. This does not save time or memory for the construction of the matrix itself, but it will lead to savings in time and memory later on, when the matrix is used.

To ensure a matrix C is iso-valued, simply use GrB_assign to compute C<C,struct>=1, or assign whatever value of scalar you wish. It is essential to use a structural mask. Otherwise, it is not clear that all entries in C will be assigned the same value. The following code takes $O(1)$ time, and it resets the size of the numerical part of the C matrix to be $O(1)$ in size:

```
bool scalar = true ;
GrB_Matrix_assign (C, C, NULL, scalar, GrB_ALL, nrows, GrB_ALL, ncols,
    GrB_DESC_S) ;
```

The MATLAB/Octave analog of the code above is C=spones(C).

The second case for where iso matrices and vectors are useful is to use them with operations that do not necessarily access all of their content. Suppose you have a matrix A of arbitrarily large dimension (say n-by-n where n=2^60, of type GrB_FP64). A matrix this large can be represented by SuiteSparse:GraphBLAS, but only in a hypersparse form.

Now, suppose you wish to compute the maximum value in each row, reducing the matrix to a vector. This can be done with GrB_reduce:

```
GrB_Vector_new (&v, GrB_FP64, n) ;
GrB_reduce (v, NULL, GrB_MAX_MONOID_FP64, A, NULL) ;
```

It can also be done with GrB_mxv, by creating an iso full vector x. The creation of x takes $O(1)$ time and memory, and the GrB_mxv computation takes $O(e)$ time (with modest assumptions; if A needs to be transposed the time would be $O(e \log e)$).

```
GrB_Vector_new (&v, GrB_FP64, n) ;
GrB_Vector_new (&x, GrB_FP64, n) ;
GrB_assign (x, NULL, NULL, 1, GrB_ALL, n, NULL) ;
GrB_mxv (v, NULL, NULL, GrB_MAX_FIRST_SEMIRING_FP64, A, x, NULL) ;
```

The above computations are identical in SuiteSparse:GraphBLAS. Internally, GrB_reduce creates x and calls GrB_mxv. Using GrB_mxm directly gives the user application additional flexibility in creating new computations that exploit the multiplicative operator in the semiring. GrB_reduce always uses the FIRST operator in its semiring, but any other binary operator can be used instead when using GrB_mxv.

15.2 Iso matrices from matrix multiplication

Consider GrB_mxm, GrB_mxv, and GrB_vxm, and let C=A*B, where no mask is present, or C<M>=A*B where C is initially empty. If C is not initially empty, then these rules apply to a temporary matrix T<M>=A*B, which is initially empty and is then assigned to C via C<M>=T.

The iso property of C is determined with the following rules, where the first rule that fits defines the property and value of C.

- If the semiring includes a index-based multiplicative operator (GxB_FIRSTI, GrB_SECONDI, and related operators), then C is never iso.
- Define an *iso-monoid* as a built-in monoid with the property that reducing a set of $n > 1$ identical values x returns the same value x. These are the MIN MAX LOR LAND BOR BAND and ANY monoids. All other monoids are not iso monoids: PLUS, TIMES, LXNOR, EQ, BXOR, BXNOR, and all user-defined monoids. Currently, there is no mechanism for telling SuiteSparse:GraphBLAS that a user-defined monoid is an iso-monoid.
- If the multiplicative op is PAIR (same as ONEB), and the monoid is an iso-monoid, or the EQ or TIMES monoids, then C is iso with a value of 1.
- If both B and the monoid are iso, and the multiplicative op is SECOND or ANY, then C is iso with a value of b.
- If both A and the monoid are iso, and the multiplicative op is FIRST or ANY, then C is iso with a value of a.

- If `A`, `B`, and the monoid are all iso, then `C` is iso, with a value $c = f(a, b)$, where f is any multiplicative op (including user-defined, which assumes that a user-defined f has no side effects).

- If `A` and `B` are both iso and full (all entries present, regardless of the format of the matrices), then `C` is iso and full. Its iso value is computed in $O(\log(n))$ time, via a reduction of n copies of the value $t = f(a, b)$ to a scalar. The storage required to represent `C` is just $O(1)$, regardless of its dimension. Technically, the `PLUS` monoid could be computed as $c = nt$ in $O(1)$ time, but the log-time reduction works for any monoid, including user-defined ones.

- Otherwise, `C` is not iso.

15.3 Iso matrices from eWiseMult and kronecker

Consider `GrB_eWiseMult`. Let `C=A.*B`, or `C<M>=A.*B` with any mask and where `C` is initially empty, where `.*` denotes a binary operator $f(x, y)$ applied with `eWiseMult`. These rules also apply to `GrB_kronecker`.

- If the operator is index-based (`GxB_FIRSTI` and related) then `C` is not iso.

- If the op is `PAIR` (same as `ONEB`), then `C` is iso with $c = 1$.

- If `B` is iso and the op is `SECOND` or `ANY`, then `C` is iso with $c = b$.

- If `A` is iso and the op is `FIRST` or `ANY`, then `C` is iso with $c = a$.

- If both `A` and `B` are iso, then `C` is iso with $c = f(a, b)$.

- Otherwise, `C` is not iso.

15.4 Iso matrices from eWiseAdd

Consider `GrB_eWiseAdd`, and also the accumulator phase of `C<M>+=T` when an accumulator operator is present. Let `C=A+B`, or `C<M>=A+B` with any mask and where `C` is initially empty.

- If both `A` and `B` are full (all entries present), then the rules for `eWiseMult` in Section 15.3 are used instead.

- If the operator is index-based (GxB_FIRSTI and related) then C is not iso.
- If a and b differ (when typecasted to the type of C), then C is not iso.
- If $c = f(a, b) = a = b$ holds, then C is iso, where $f(a, b)$ is the operator.
- Otherwise, C is not iso.

15.5 Iso matrices from eWiseUnion

GxB_eWiseUnion is very similar to GrB_eWiseAdd, but the rules for when the result is iso-valued are very different.

- If both A and B are full (all entries present), then the rules for eWiseMult in Section 15.3 are used instead.
- If the operator is index-based (GxB_FIRSTI and related) then C is not iso.
- If the op is PAIR (same as ONEB), then C is iso with $c = 1$.
- If B is iso and the op is SECOND or ANY, and the input scalar beta matches b (the iso-value of B), then C is iso with $c = b$.
- If A is iso and the op is FIRST or ANY, and the input scalar alpha matches a (the iso-value of A), then C is iso with $c = a$.
- If both A and B are iso, and $f(a, b) = f(\alpha, b) = f(a, \beta)$, then C is iso with $c = f(a, b)$.
- Otherwise, C is not iso.

15.6 Reducing iso matrices to a scalar or vector

If A is iso with e entries, reducing it to a scalar takes $O(\log(e))$ time, regardless of the monoid used to reduce the matrix to a scalar. Reducing A to a vector c is the same as the matrix-vector multiply c=A*x or c=A'*x, depending on the descriptor, where x is an iso full vector (refer to Section 15.2).

15.7 Iso matrices from apply

Let `C=f(A)` denote the application of a unary operator `f`, and let `C=f(A,s)` and `C=f(s,A)` denote the application of a binary operator with `s` a scalar.

- If the operator is index-based (`GxB_POSITION*`, `GxB_FIRSTI`, and related) then `C` is not iso.
- If the operator is `ONE` or `PAIR` (same as `ONEB`), then `C` iso with $c = 1$.
- If the operator is `FIRST` or `ANY` with `C=f(s,A)`, then `C` iso with $c = s$.
- If the operator is `SECOND` or `ANY` with `C=f(A,s)`, then `C` iso with $c = s$.
- If `A` is iso then `C` is iso, with the following value of c:
  - If the op is `IDENTITY`, then $c = a$.
  - If the op is unary with `C=f(A)`, then $c = f(a)$.
  - If the op is binary with `C=f(s,A)`, then $c = f(s, a)$.
  - If the op is binary with `C=f(A,s)`, then $c = f(a, s)$.
- Otherwise, `C` is not iso.

15.8 Iso matrices from select

Let `C=select(A)` denote the application of a `GrB_IndexUnaryOp` operator in `GrB_select`.

- If `A` is iso, then `C` is iso with $c = a$.
- If the operator is any `GrB_VALUE*_BOOL` operator, with no typecasting, and the test is true only for a single boolean value, then `C` is iso.
- If the operator is `GrB_VALUEEQ_*`, with no typecasting, then `C` is iso, with $c = t$ where t is the value of the scalar `y`.
- If the operator is `GrB_VALUELE_UINT*`, with no typecasting, and the scalar `y` is zero, then `C` is iso with $c = 0$.
- Otherwise, `C` is not iso.

15.9 Iso matrices from assign and subassign

These rules are somewhat complex. Consider the assignment C<M>(I,J)=... with GrB_assign. Internally, this assignment is converted into C(I,J)<M(I,J)>=... and then GxB_subassign is used. Thus, all of the rules below assume the form C(I,J)<M>=... where M has the same size as the submatrix C(I,J).

15.9.1 Assignment with no accumulator operator

If no accumulator operator is present, the following rules are used.

- For matrix assignment, A must be iso. For scalar assignment, the single scalar is implicitly expanded into an iso matrix A of the right size. If these rules do not hold, C is not iso.

- If A is not iso, or if C is not iso on input, then C is not iso on output.

- If C is iso or empty on input, and A is iso (or scalar assignment is begin performed) and the iso values c and a (or the scalar s) match, then the following forms of assignment result in an iso matrix C on output:

  - C(I,J) = scalar
  - C(I,J)<M> = scalar
  - C(I,J)<!M> = scalar
  - C(I,J)<M,replace> = scalar
  - C(I,J)<!M,replace> = scalar
  - C(I,J) = A
  - C(I,J)<M> = A
  - C(I,J)<!M> = A
  - C(I,J)<M,replace> = A
  - C(I,J)<!M,replace> = A

- For these forms of assignment, C is always iso on output, regardless of its iso property on input:

  - C = scalar

  - C<M,struct>=scalar; C empty on input.
  - C<C,struct>=scalar

- For these forms of assignment, C is always iso on output if A is iso:
  - C = A
  - C<M,str> = A; C empty on input.

15.9.2 Assignment with an accumulator operator

If an accumulator operator is present, the following rules are used. Index-based operators (GxB_FIRSTI and related) cannot be used as accumulator operators, so these rules do not consider that case.

- For matrix assignment, A must be iso. For scalar assignment, the single scalar is implicitly expanded into an iso matrix A of the right size. If these rules do not hold, C is not iso.
- For these forms of assignment C is iso if C is empty on input, or if $c = c + a$ for the where a is the iso value of A or the value of the scalar for scalar assignment.
  - C(I,J) += scalar
  - C(I,J)<M> += scalar
  - C(I,J)<!M> += scalar
  - C(I,J)<M,replace> += scalar
  - C(I,J)<!M,replace> += scalar
  - C(I,J)<M,replace> += A
  - C(I,J)<!M,replace> += A
  - C(I,J) += A
  - C(I,J)<M> += A
  - C(I,J)<!M> += A
  - C += A

15.10 Iso matrices from build methods

GxB_Matrix_build_Scalar and GxB_Vector_build_Scalar always construct an iso matrix/vector.

GrB_Matrix_build and GrB_Vector_build can also construct iso matrices and vectors. A non-iso matrix/vector is constructed first, and then the entries are checked to see if they are all equal. The resulting iso-valued matrix/vector will be efficient to use and will use less memory than a non-iso matrix/vector. However, constructing an iso matrix/vector with GrB_Matrix_build and GrB_Vector_build will take more time and memory than constructing the matrix/vector with GxB_Matrix_build_Scalar or GxB_Vector_build_Scalar.

15.11 Iso matrices from other methods

- For GrB_Matrix_dup and GrB_Vector_dup, the output matrix/vector has the same iso property as the input matrix/vector.
- GrB_*_setElement_* preserves the iso property of the matrix/vector it modifies, if the input scalar is equal to the iso value of the matrix/vector. If the matrix or vector has no entries, the first call to setElement makes it iso. This allows a sequence of setElement calls with the same scalar value to create an entire iso matrix or vector, if starting from an empty matrix or vector.
- GxB_Matrix_concat constructs an iso matrix as its result if all input tiles are either empty or iso.
- GxB_Matrix_split constructs its output tiles as iso if its input matrix is iso.
- GxB_Matrix_diag and GrB_Matrix_diag construct an iso matrix if its input vector is iso.
- GxB_Vector_diag constructs an iso vector if its input matrix is iso.
- GrB_*extract constructs an iso matrix/vector if its input matrix/vector is iso.
- GrB_transpose constructs an iso matrix if its input is iso.
- The GxB_Container methods preserve the iso property of their matrices/vectors.

15.12 Iso matrices not exploited

There are many cases where an matrix may have the iso property but it is not detected by SuiteSparse:GraphBLAS. For example, if `A` is non-iso, `C=A(I,J)` from `GrB_extract` may be iso, if all entries in the extracted submatrix have the same value. Future versions of SuiteSparse:GraphBLAS may extend the rules described in this section to detect these cases.

16 Performance

Getting the best performance out of an algorithm that relies on GraphBLAS can depend on many factors. This section describes some of the possible performance pitfalls you can hit when using SuiteSparse:GraphBLAS, and how to avoid them (or at least know when you've encountered them).

16.1 The burble is your friend

Turn on the burble with `GrB_set (GrB_GLOBAL, true, GxB_BURBLE)`. You will get a single line of output from each (significant) call to GraphBLAS. The burble output can help you detect when you are likely using sub-optimal methods, as described in the next sections. When the JIT is in use the burble reports when a JIT kernel is run (which is quick), loaded for the first time (which takes a small amount of time), and when a JIT kernels is compiled (which can take a few tenths of a second or more). The compiler command is printed in full. If you encounter a compiler error, you can cut-and-paste the compiler command while outside of your application to help track down the compiler error.

16.2 Data types and typecasting: use the JIT

If the JIT is disabled, avoid mixing data types and relying on typecasting as much as possible. SuiteSparse:GraphBLAS has a set of highly-tuned kernels for each data type, and many operators and semirings, but there are too many combinations to generate ahead of time. If typecasting is required, or if SuiteSparse:GraphBLAS does not have a kernel for the specific operator or semiring, the word `generic` will appear in the burble. The generic methods rely on function pointers for each operation on every scalar, so they are slow. Enabling the JIT avoids this problem, since GraphBLAS can then compile kernel specific to the types used.

Without the JIT, the only time that typecasting is fast is when computing `C=A` via `GrB_assign` or `GrB_apply`, where the data types of `C` and `A` can differ. In this case, one of $13^2 = 169$ kernels are called, each of which performs the specific typecasting requested, without relying on function pointers.

16.3 Matrix data structures: sparse, hypersparse, bitmap, or full

SuiteSparse:GraphBLAS tries to automatically determine the best data structure for your matrices and vectors, selecting between sparse, hypersparse, bitmap, and full formats. By default, all 4 formats can be used. A matrix typically starts out hypersparse when it is created by `GrB_Matrix_new`, and then changes during its lifetime, possibly taking on all four different formats at different times. This can be modified via `GrB_set`. For example, this line of code:

```
GrB_set (A, GxB_SPARSE + GxB_BITMAP, GxB_SPARSITY_CONTROL) ;
```

tells SuiteSparse that the matrix `A` can be held in either sparse or bitmap format (at its discretion), but not hypersparse or full. The bitmap format will be used if the matrix has enough entries, or sparse otherwise. Sometimes this selection is best controlled by the user algorithm, so a single format can be requested:

```
GrB_set (A, GxB_SPARSE, GxB_SPARSITY_CONTROL) ;
```

This ensures that SuiteSparse will primarily use the sparse format. This is still just a hint, however. The data structure is opaque and SuiteSparse is free to choose otherwise. In particular, if you insist on using only the `GxB_FULL` format, then that format is used when all entries are present. However, if the matrix is not actually full with all entries present, then the bitmap format is used instead. The full format does not preserve the sparsity structure in this case. Any GraphBLAS library must preserve the proper structure, per the C Specification. This is critical in a graph algorithm, since an edge (i,j) of weight zero, say, is not the same as no edge (i,j) at all.

16.4 Matrix formats: by row or by column, or using the transpose of a matrix

By default, SuiteSparse uses a simple rule: all matrices are held by row, unless the consist of a single column, in which case they are held by column. All vectors are treated as if they are n-by-1 matrices with a single column. Changing formats from row-oriented to column-oriented can have significant performance implications, so SuiteSparse never tries to outguess the application. It just uses this simple rule.

However, there are cases where changing the format can greatly improve performance. There are two ways to handle this, which in the end are equivalent in the SuiteSparse internals. You can change the format (row to column oriented, or visa versa), or work with the explicit transpose of a matrix in the same storage orientation.

There are cases where SuiteSparse must explicitly transpose an input matrix, or the output matrix, in order to perform a computation. For example, if all matrices are held in row-oriented fashion, SuiteSparse does not have a method for computing `C=A'*B`, where `A` is transposed. Thus, SuiteSparse either computes a temporary transpose of its input matrix `AT=A` and then `C=AT*B`, or it swaps the computations, performing `C=(B'*A)'`, which requires an explicit transpose of `BT=B`, and a transpose of the final result to obtain `C`.

These temporary transposes are costly to compute, taking time and memory. They are not kept, but are discarded when the method returns to the user application. If you see the term `transpose` in the burble output, and if you need to perform this computation many times, try constructing your own explicit transpose, say `AT=A'`, via `GrB_transpose`, or create a copy of `A` but held in another orientation via `GrB_set`. For example, assuming the default matrix format is by-row, and that `A` is `m`-by-`n` of type `GrB_FP32`:

```
// method 1: AT = A'
GrB_Matrix_new (AT, GrB_FP32, n, m) ;
GrB_transpose (AT, NULL, NULL, A, NULL) ;

// method 2: A2 = A but held by column instead of by row
// note: doing the set before the assign is faster than the reverse
GrB_Matrix_new (A2, GrB_FP32, m, n) ;
GrB_set (A2, GrB_COLMAJOR, GrB_STORAGE_ORIENTATION_HINT) ;
GrB_assign (A2, NULL, NULL, A, GrB_ALL, m, GrB_ALL, n, NULL) ;
```

Internally, the data structure for `AT` and `A2` are nearly identical (that is, the tranpose of `A` held in row format is the same as `A` held in column format). Using either of them in subsequent calls to GraphBLAS will allow SuiteSparse to avoid computing an explicit transpose. The two matrices `AT` and `A2` do differ in one very significant way: their dimensions are different, and they behave differement mathematically. Computing `C=A'*B` using these matrices would differ:

```
// method 1: C=A'*B using AT
GrB_mxm (C, NULL, NULL, semiring, AT, B, NULL) ;

// method 2: C=A'*B using A2
GrB_mxm (C, NULL, NULL, semiring, A2, B, GrB_DESC_T0) ;
```

The first method computes `C=AT*B`. The second method computes `C=A2'*B`, but the result of both computations is the same, and internally the same kernels will be used.

16.5 Push/pull optimization

Closely related to the discussion above on when to use a matrix or its transpose is the exploitation of "push/pull" direction optimization. In linear algebraic terms, this is simply deciding whether to multiply by the matrix or its transpose. Examples can be see in the BFS and Betweeness-Centrality methods of LAGraph. Here is the BFS kernel:

```
int sparsity = do_push ? GxB_SPARSE : GxB_BITMAP ;
GrB_set (q, sparsity, GxB_SPARSITY_CONTROL) ;
if (do_push)
{
    // q'{!pi} = q'*A
    GrB_vxm (q, pi, NULL, semiring, q, A, GrB_DESC_RSC) ;
}
else
{
    // q{!pi} = AT*q
    GrB_mxv (q, pi, NULL, semiring, AT, q, GrB_DESC_RSC) ;
}
```

The call to `GrB_set` is optional, since SuiteSparse will likely already determine that a bitmap format will work best when the frontier `q` has many entries, which is also when the pull step is fastest. The push step relies on a sparse vector times sparse matrix method originally due to Gustavson. The output is computed as a set union of all rows `A(i,:)` where `q(i)` is present on input. This set union is very fast when `q` is very sparse. The pull step relies on a sequence of dot product computations, one per possible entry in the output `q`, and it uses the matrix `AT` which is a row-oriented copy of the explicit transpose of the adjacency matrix `A`.

Mathematically, the results of the two methods are identical, but internally, the data format of the input matrices is very different (using `A` held by row, or AT held by row which is the same as a copy of A that is held by column), and the algorithms used are very different.

16.6 Computing with full matrices and vectors

Sometimes the best approach to getting the highest performance is to use dense vectors, and occassionaly dense matrices are tall-and-thin or short-and-fat. Packages such as Julia, Octave, or MATLAB, when dealing with the conventional plus-times semirings, assume that multiplying a sparse matrix A times a dense vector x, y=A*x, will result in a dense vector y. This is not always the case, however. GraphBLAS must always return a result that respects the sparsity structure of the output matrix or vector. If the ith row of A has no entries then y(i) must not appear as an entry in the vector y, so it cannot be held as a full vector. As a result, the following computation can be slower than it could be:

```
GrB_mxv (y, NULL, NULL, semiring, A, x, NULL) ;
```

SuiteSparse must do extra work to compute the sparsity of this vector y, but if this is not needed, and y can be padded with zeros (or the identity value of the monoid, to be precise), a faster method can be used, by relying on the accumulator. Instead of computing y=A*x, set all entries of y to zero first, and then compute y+=A*x where the accumulator operator and type matches the monoid of the semiring. SuiteSparse has special kernels for this case; you can see them in the burble as F+=S*F for example.

```
// y = 0
GrB_assign (y, NULL, NULL, 0, GrB_ALL, n, NULL) ;
// y += A*x
GrB_mxv (y, NULL, GrB_PLUS_FP32, GrB_PLUS_TIMES_SEMIRING_FP32, A, x, NULL) ;
```

You can see this computation in the LAGraph PageRank method, where all entries of r are set to the teleport scalar first.

```
for (iters = 0 ; iters < itermax && rdiff > tol ; iters++)
{
    // swap t and r ; now t is the old score
    GrB_Vector temp = t ; t = r ; r = temp ;
    // w = t ./ d
    GrB_eWiseMult (w, NULL, NULL, GrB_DIV_FP32, t, d, NULL) ;
    // r = teleport
```

```
    GrB_assign (r, NULL, NULL, teleport, GrB_ALL, n, NULL) ;
    // r += A'*w
    GrB_mxv (r, NULL, GrB_PLUS_FP32, LAGraph_plus_second_fp32, AT, w, NULL) ;
    // t -= r
    GrB_assign (t, NULL, GrB_MINUS_FP32, r, GrB_ALL, n, NULL) ;
    // t = abs (t)
    GrB_apply (t, NULL, NULL, GrB_ABS_FP32, t, NULL) ;
    // rdiff = sum (t)
    GrB_reduce (&rdiff, NULL, GrB_PLUS_MONOID_FP32, t, NULL) ;
}
```

SuiteSparse exploits the iso-valued property of the scalar-to-vector assignment of `y=0`, or `r=teleport`, and performs these assignments in O(1) time and space. Because the `r` vector start out as full on input to `GrB_mxv`, and because there is an accumulatr with no mask, no entries in the input/output vector `r` will be deleted, even if `A` has empty rows. The call to `GrB_mxv` exploits this, and is able to use a fast kernel for this computation. SuiteSparse does not need to compute the sparsity pattern of the vector `r`.

16.7 Iso-valued matrices and vectors

Using iso-valued matrices and vectors is always faster than using matrices and vectors whose entries can have different values. Iso-valued matrices are very important in graph algorithms. For example, an unweighted graph is best represented as an iso-valued sparse matrix, and unweighted graphs are very common. The burble output, `GxB_print`, or `GrB_get` can all be used to report whether or not your matrix or vector is iso-valued.

Sometimes a matrix or vector may have values that are all the same, but SuiteSparse hasn't detected this. If this occurs, you can force a matrix or vector to be iso-valued by assigning a single scalar to all its entries.

```
// C<s(C)> = 3.14159
GrB_assign (C, C, NULL, 3.14159, GrB_ALL, m, GrB_ALL, n, GrB_DESC_S) ;
```

The matrix `C` is used as its own mask. The descriptor is essential here, telling the mask to be used in a structural sense, without regard to the values of the entries in the mask. This assignment sets all entries that already exist in `C` to be equal to a single value, 3.14159. The sparsity structure of `C` does not change. Of course, any scalar can be used; the value 1 is common for unweighted graphs. SuiteSparse:GraphBLAS performs the above assignment in O(1) time and space, independent of the dimension of `C` or the number of entries in contains.

16.8 User-defined types and operators: use the JIT

If the JIT is disabled, these will be slow. With the JIT enabled, data types and operators are just as fast as built-in types and operators. A CUDA JIT for the GPU is in progress, collaboration with Joe Eaton and Corey Nolet. A SYCL/OpenCL JIT is under consideration, but work has not yet been started.

16.9 About NUMA systems

I have tested this package extensively on multicore single-socket systems, but have not yet optimized it for multi-socket systems with a NUMA architecture. That will be done in a future release. If you publish benchmarks with this package, please state the SuiteSparse:GraphBLAS version, and a caveat if appropriate. If you see significant performance issues when going from a single-socket to multi-socket system, I would like to hear from you so I can look into it.

17 Examples

Several examples of how to use GraphBLAS are listed below. They all appear in the `Demo` folder of SuiteSparse:GraphBLAS. Programs in the `Demo` folder are meant as simple examples; for the fastest methods, see LAgraph (Section 17.1).

1. creating a random matrix
2. creating a finite-element matrix
3. reading a matrix from a file
4. complex numbers as a user-defined type
5. matrix import/export

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

17.1 LAGraph

The LAGraph project is a community-wide effort to create graph algorithms based on GraphBLAS (any implementation of the API, not just SuiteSparse: GraphBLAS). Some of the algorithms and utilities in LAGraph are listed in the table below. Many additional algorithms are planned. Refer to https://github.com/GraphBLAS/LAGraph for a current list of algorithms. All functions in the `Demo/` folder in SuiteSparse:GraphBLAS will eventually be translated into algorithms or utilities for LAGraph, and then removed from `GraphBLAS/Demo`.

To use LAGraph with SuiteSparse:GraphBLAS, place the two folders `LAGraph` and `GraphBLAS` in the same parent directory. This allows the `cmake` script in LAGraph to find the copy of GraphBLAS. Alternatively, the GraphBLAS source could be placed anywhere, as long as `sudo make install` is performed.

17.2 Creating a random matrix

The `random_matrix` function in the `Demo` folder generates a random matrix with a specified dimension and number of entries, either symmetric or unsymmetric, and with or without self-edges (diagonal entries in the matrix).

It relies on simple_rand* functions in the Demo folder to provide a portable random number generator that creates the same sequence on any computer and operating system.

random_matrix can use one of two methods: GrB_Matrix_setElement and GrB_Matrix_build. The former method is very simple to use:

```
GrB_Matrix_new (&A, GrB_FP64, nrows, ncols) ;
for (int64_t k = 0 ; k < ntuples ; k++)
{
    GrB_Index i = simple_rand_i ( ) % nrows ;
    GrB_Index j = simple_rand_i ( ) % ncols ;
    if (no_self_edges && (i == j)) continue ;
    double x = simple_rand_x ( ) ;
    // A (i,j) = x
    GrB_Matrix_setElement (A, x, i, j) ;
    if (make_symmetric)
    {
        // A (j,i) = x
        GrB_Matrix_setElement (A, x, j, i) ;
    }
}
```

The above code can generate a million-by-million sparse double matrix with 200 million entries in 66 seconds (6 seconds of which is the time to generate the random i, j, and x), including the time to finish all pending computations. The user application does not need to create a list of all the tuples, nor does it need to know how many entries will appear in the matrix. It just starts from an empty matrix and adds them one at a time in arbitrary order. GraphBLAS handles the rest. This method is not feasible in MATLAB.

The next method uses GrB_Matrix_build. It is more complex to use than setElement since it requires the user application to allocate and fill the tuple lists, and it requires knowledge of how many entries will appear in the matrix, or at least a good upper bound, before the matrix is constructed. It is slightly faster, creating the same matrix in 60 seconds, 51 seconds of which is spent in GrB_Matrix_build.

```
GrB_Index *I, *J ;
double *X ;
int64_t s = ((make_symmetric) ? 2 : 1) * nedges + 1 ;
I = malloc (s * sizeof (GrB_Index)) ;
J = malloc (s * sizeof (GrB_Index)) ;
```

```
X = malloc (s * sizeof (double   )) ;
if (I == NULL || J == NULL || X == NULL)
{
    // out of memory
    if (I != NULL) free (I) ;
    if (J != NULL) free (J) ;
    if (X != NULL) free (X) ;
    return (GrB_OUT_OF_MEMORY) ;
}
int64_t ntuples = 0 ;
for (int64_t k = 0 ; k < nedges ; k++)
{
    GrB_Index i = simple_rand_i ( ) % nrows ;
    GrB_Index j = simple_rand_i ( ) % ncols ;
    if (no_self_edges && (i == j)) continue ;
    double x = simple_rand_x ( ) ;
    // A (i,j) = x
    I [ntuples] = i ;
    J [ntuples] = j ;
    X [ntuples] = x ;
    ntuples++ ;
    if (make_symmetric)
    {
        // A (j,i) = x
        I [ntuples] = j ;
        J [ntuples] = i ;
        X [ntuples] = x ;
        ntuples++ ;
    }
}
GrB_Matrix_build (A, I, J, X, ntuples, GrB_SECOND_FP64) ;
```

The equivalent `sprandsym` function in MATLAB takes 150 seconds, but `sprandsym` uses a much higher-quality random number generator to create the tuples `[I,J,X]`. Considering just the time for `sparse(I,J,X,n,n)` in `sprandsym` (equivalent to `GrB_Matrix_build`), the time is 70 seconds. That is, each of these three methods, `setElement` and `build` in SuiteSparse:GraphBLAS, and `sparse` in MATLAB, are equally fast.

17.3 Creating a finite-element matrix

Suppose a finite-element matrix is being constructed, with `k=40,000` finite-element matrices, each of size `8`-by-`8`. The following operations (in pseudo-

MATLAB notation) are very efficient in SuiteSparse:GraphBLAS.

```
A = sparse (m,n) ; % create an empty n-by-n sparse GraphBLAS matrix
for i = 1:k
    construct a 8-by-8 sparse or dense finite-element F
    I and J define where the matrix F is to be added:
    I = a list of 8 row indices
    J = a list of 8 column indices
    % using GrB_assign, with the 'plus' accum operator:
    A (I,J) = A (I,J) + F
end
```

If this were done in MATLAB or in GraphBLAS with blocking mode enabled, the computations would be extremely slow. A far better approach is to construct a list of tuples `[I,J,X]` and to use `sparse(I,J,X,n,n)`. This is identical to creating the same list of tuples in GraphBLAS and using the `GrB_Matrix_build`, which is equally fast.

In SuiteSparse:GraphBLAS, the performance of both methods is essentially identical, and roughly as fast as `sparse` in MATLAB. Inside SuiteSparse:GraphBLAS, `GrB_assign` is doing the same thing. When performing `A(I,J)=A(I,J)+F`, if it finds that it cannot quickly insert an update into the `A` matrix, it creates a list of pending tuples to be assembled later on. When the matrix is ready for use in a subsequent GraphBLAS operation (one that normally cannot use a matrix with pending computations), the tuples are assembled all at once via `GrB_Matrix_build`.

GraphBLAS operations on other matrices have no effect on when the pending updates of a matrix are completed. Thus, any GraphBLAS method or operation can be used to construct the `F` matrix in the example above, without affecting when the pending updates to `A` are completed.

The MATLAB `wathen.m` script is part of Higham's `gallery` of matrices [Hig02]. It creates a finite-element matrix with random coefficients for a 2D mesh of size `nx`-by-`ny`, a matrix formulation by Wathen [Wat87]. The pattern of the matrix is fixed; just the values are randomized. The GraphBLAS equivalent can use either `GrB_Matrix_build`, or `GrB_assign`. Both methods have good performance. The `GrB_Matrix_build` version below is about 15% to 20% faster than the MATLAB `wathen.m` function, regardless of the problem size. It uses the identical algorithm as `wathen.m`.

```
int64_t ntriplets = nx*ny*64 ;
I = malloc (ntriplets * sizeof (int64_t)) ;
```

```
    J = malloc (ntriplets * sizeof (int64_t)) ;
    X = malloc (ntriplets * sizeof (double )) ;
    if (I == NULL || J == NULL || X == NULL)
    {
        FREE_ALL ;
        return (GrB_OUT_OF_MEMORY) ;
    }
    ntriplets = 0 ;
    for (int j = 1 ; j <= ny ; j++)
    {
        for (int i = 1 ; i <= nx ; i++)
        {
            nn [0] = 3*j*nx + 2*i + 2*j + 1 ;
            nn [1] = nn [0] - 1 ;
            nn [2] = nn [1] - 1 ;
            nn [3] = (3*j-1)*nx + 2*j + i - 1 ;
            nn [4] = 3*(j-1)*nx + 2*i + 2*j - 3 ;
            nn [5] = nn [4] + 1 ;
            nn [6] = nn [5] + 1 ;
            nn [7] = nn [3] + 1 ;
            for (int krow = 0 ; krow < 8 ; krow++) nn [krow]-- ;
            for (int krow = 0 ; krow < 8 ; krow++)
            {
                for (int kcol = 0 ; kcol < 8 ; kcol++)
                {
                    I [ntriplets] = nn [krow] ;
                    J [ntriplets] = nn [kcol] ;
                    X [ntriplets] = em (krow,kcol) ;
                    ntriplets++ ;
                }
            }
        }
    }
    // A = sparse (I,J,X,n,n) ;
    GrB_Matrix_build (A, I, J, X, ntriplets, GrB_PLUS_FP64) ;
```

The `GrB_assign` version has the advantage of not requiring the user application to construct the tuple list, and is almost as fast as using `GrB_Matrix_build`. The code is more elegant than either the MATLAB `wathen.m` function or its GraphBLAS equivalent above. Its performance is comparable with the other two methods, but slightly slower, being about 5% slower than the MATLAB `wathen`, and 20% slower than the GraphBLAS method above.

```
    GrB_Matrix_new (&F, GrB_FP64, 8, 8) ;
```

```
for (int j = 1 ; j <= ny ; j++)
{
    for (int i = 1 ; i <= nx ; i++)
    {
        nn [0] = 3*j*nx + 2*i + 2*j + 1 ;
        nn [1] = nn [0] - 1 ;
        nn [2] = nn [1] - 1 ;
        nn [3] = (3*j-1)*nx + 2*j + i - 1 ;
        nn [4] = 3*(j-1)*nx + 2*i + 2*j - 3 ;
        nn [5] = nn [4] + 1 ;
        nn [6] = nn [5] + 1 ;
        nn [7] = nn [3] + 1 ;
        for (int krow = 0 ; krow < 8 ; krow++) nn [krow]-- ;
        for (int krow = 0 ; krow < 8 ; krow++)
        {
            for (int kcol = 0 ; kcol < 8 ; kcol++)
            {
                // F (krow,kcol) = em (krow, kcol)
                GrB_Matrix_setElement (F, em (krow,kcol), krow, kcol) ;
            }
        }
        // A (nn,nn) += F
        GrB_assign (A, NULL, GrB_PLUS_FP64, F, nn, 8, nn, 8, NULL) ;
    }
}
```

Since there is no `Mask`, and since `GrB_REPLACE` is not used, the call to `GrB_assign` in the example above is identical to `GxB_subassign`. Either one can be used, and their performance would be identical.

Refer to the `wathen.c` function in the `Demo` folder, which uses GraphBLAS to implement the two methods above, and two additional ones.

17.4 Reading a matrix from a file

See also `LAGraph_mmread` and `LAGraph_mmwrite`, which can read and write any matrix in Matrix Market format, and `LAGraph_binread` and `LAGraph_binwrite`, which read/write a matrix from a binary file. The binary file I/O functions are much faster than the `read_matrix` function described here, and also much faster than `LAGraph_mmread` and `LAGraph_mmwrite`.

The `read_matrix` function in the `Demo` reads in a triplet matrix from a file, one line per entry, and then uses `GrB_Matrix_build` to create the matrix. It creates a second copy with `GrB_Matrix_setElement`, just to test that method and compare the run times. Section 17.2 has already compared `build` versus `setElement`.

The function can return the matrix as-is, which may be rectangular or unsymmetric. If an input parameter is set to make the matrix symmetric, `read_matrix` computes `A=(A+A')/2` if `A` is square (turning all directed edges into undirected ones). If `A` is rectangular, it creates a bipartite graph, which is the same as the augmented matrix, `A = [0 A ; A' 0]`. If `C` is an `n`-by-`n` matrix, then `C=(C+C')/2` can be computed as follows in GraphBLAS, (the `scale2` function divides an entry by 2):

```
GrB_Descriptor_new (&dt2) ;
GrB_set (dt2, GrB_TRAN, GrB_INP1) ;
GrB_Matrix_new (&A, GrB_FP64, n, n) ;
GrB_eWiseAdd (A, NULL, NULL, GrB_PLUS_FP64, C, C, dt2) ;    // A=C+C'
GrB_free (&C) ;
GrB_Matrix_new (&C, GrB_FP64, n, n) ;
GrB_UnaryOp_new (&scale2_op, scale2, GrB_FP64, GrB_FP64) ;
GrB_apply (C, NULL, NULL, scale2_op, A, NULL) ;             // C=A/2
GrB_free (&A) ;
GrB_free (&scale2_op) ;
```

This is of course not nearly as elegant as `A=(A+A')/2` in MATLAB, but with minor changes it can work on any type and use any built-in operators instead of `PLUS`, or it can use any user-defined operators and types. The above code in SuiteSparse:GraphBLAS takes 0.60 seconds for the `Freescale2` matrix, slightly slower than MATLAB (0.55 seconds).

Constructing the augmented system is more complicated using the GraphBLAS C API Specification since it does not yet have a simple way of specifying a range of row and column indices, as in `A(10:20,30:50)` in MATLAB (`GxB_RANGE` is a SuiteSparse:GraphBLAS extension that is not in the Specification). Using the C API in the Specification, the application must instead build a list of indices first, `I=[10, 11 ... 20]`.

Thus, to compute the MATLAB equivalent of `A = [0 A ; A' 0]`, index lists `I` and `J` must first be constructed:

```
int64_t n = nrows + ncols ;
I = malloc (nrows * sizeof (int64_t)) ;
J = malloc (ncols * sizeof (int64_t)) ;
// I = 0:nrows-1
// J = nrows:n-1
if (I == NULL || J == NULL)
{
```

```
        if (I != NULL) free (I) ;
        if (J != NULL) free (J) ;
        return (GrB_OUT_OF_MEMORY) ;
    }
    for (int64_t k = 0 ; k < nrows ; k++) I [k] = k ;
    for (int64_t k = 0 ; k < ncols ; k++) J [k] = k + nrows ;
```

Once the index lists are generated, however, the resulting GraphBLAS operations are fairly straightforward, computing `A=[0 C ; C' 0]`.

```
    GrB_Descriptor_new (&dt1) ;
    GrB_set (dt1, GrB_TRAN, GrB_INP0) ;
    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). The `GxB_Matrix_concat` function would be faster still (this example was written prior to `GxB_Matrix_concat` was added to SuiteSparse:GraphBLAS).

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.

17.5 User-defined types and operators

The `Demo` folder contains two working examples of user-defined types, first discussed in Section 6.1.1: `double complex`, and a user-defined `typedef` called `wildtype` with a `struct` containing a string and a 4-by-4 `float` matrix.

Double Complex: Prior to v3.3, GraphBLAS did not have a native complex type. It now appears as the `GxB_FC64` predefined type, but a complex type can also easily added as a user-defined type. The `Complex_init` function in the `usercomplex.c` file in the `Demo` folder creates the `Complex` type based on the 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/Octave equivalent	description
`Complex_complex`	$D \times D \rightarrow C$	`z=complex(x,y)`	complex from real and imag.
`Complex_conj`	$C \rightarrow C$	`z=conj(x)`	complex conjugate
`Complex_real`	$C \rightarrow D$	`z=real(x)`	real part
`Complex_imag`	$C \rightarrow D$	`z=imag(x)`	imaginary part
`Complex_angle`	$C \rightarrow D$	`z=angle(x)`	phase angle
`Complex_complex_real`	$D \rightarrow C$	`z=complex(x,0)`	real to complex real
`Complex_complex_imag`	$D \rightarrow C$	`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.

17.6 User applications using OpenMP or other threading models

An example demo program (`context_demo`) is included that illustrates how a multi-threaded user application can use GraphBLAS, where each user thread calls GraphBLAS simultaneously, with nested parallelism.

GraphBLAS can also be combined with user applications that rely on MPI, the Intel TBB threading library, POSIX pthreads, Microsoft Windows threads, or any other threading library. If GraphBLAS itself is compiled with OpenMP, it will be thread safe when combined with other libraries. See Section 10.2.2 for thread-safety issues that can occur if GraphBLAS is compiled without OpenMP.

18 Compiling and Installing SuiteSparse:GraphBLAS

18.1 Quick Start

GraphBLAS requires `cmake` version 3.20 or later. It optionally can use OpenMP for best performance. For OpenMP on the Mac, see Section 18.3.6. Without OpenMP, GraphBLAS will be significantly slower since it is a highly parallel package.

A `cmake` build system is available for Linux, Mac, and Windows. For Linux or Mac, a simple Makefile wrapper is available that accesses this cmake build system. Simply do:

```
make
sudo make install
```

For Windows, open CMake and use the provided `CMakeLists.txt` file to build GraphBLAS in the `GraphBLAS/build` folder.

Next, try the demos with `make demos`. The output of the demos will be compared with expected output files in `Demo/Output`.[2]

18.2 Quick Start for MATLAB/Octave

As of GraphBLAS 9.2.0, a new and simpler method for compiling GraphBLAS and its MATLAB interface has been added. In the MATLAB/Octave Command Window, simply type:

```
cd GraphBLAS/GraphBLAS
graphblas_install
```

This will use `cmake` to compile the GraphBLAS library. You can skip the details in the remainder of this section. Next, add your `GraphBLAS/GraphBLAS` folder to your path, by editting your `startup.m` script (usually in your `Documents/MATLAB` folder). Add this line:

[2]NOTE: DO NOT publish benchmarks of these demos, and do not link against the demo library in any user application. These codes are sometimes slow, and are meant as simple illustrations only, not for performance. The fastest methods are in LAGraph, not in GraphBLAS/Demo. Benchmark LAGraph instead. Eventually, all GraphBLAS/Demos methods will be removed, and LAGraph will serve all uses: for illustration, benchmarking, and production uses.

```
addpath ('/home/me/GraphBLAS/GraphBLAS') ;
```

where `/home/me/GraphBLAS` is the top-level folder containing your copy of GraphBLAS.

The `graphblas_install` MATLAB script may fail to run `cmake`. If it does, it will print the following workaround, where the commands it tells you to use will differ depending on the platform:

```
Building GraphBLAS with cmake failed.  Try this outside of MATLAB:

   cd /home/me/GraphBLAS/GraphBLAS/build
   cmake  ..
   cmake --build . --config Release -j40

Then do this inside MATLAB:

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

where `/home/me/GraphBLAS` is your copy of GraphBLAS.

18.3 More details

18.3.1 On Linux and Mac

GraphBLAS makes extensive use of features in the C11 standard, and thus a C compiler supporting this version of the C standard is required to use all features of GraphBLAS.

Any version of the Intel `icx` compiler is highly recommended. In most cases, the Intel `icx` and the Intel OpenMP library (`libiomp`) result in the best performance. The `gcc` and the GNU OpenMP library (`libgomp`) generally gives good performance: typically on par with icx but in a few special cases significantly slower. The Intel `icc` compiler is not recommended; it results in poor performance for `#pragma omp atomic`.

If you are using a C compiler that does not support the C11 standard, such as `cl` in Microsoft Visual Studio, then the `_Generic` keyword is not available. SuiteSparse:GraphBLAS will still compile, but you will not have access to polymorphic functions such as `GrB_assign`. You will need to use the non-polymorphic functions instead.

To compile SuiteSparse:GraphBLAS, simply type `make` in the main GraphBLAS folder, which compiles the library with your default system compiler. This compile GraphBLAS using 8 threads, which will take a long time. To compile with more threads (40, for this example), use:

```
make JOBS=40
```

To use a non-default compiler with 4 threads:

```
make CC=icx CXX=icpx JOBS=4
```

GraphBLAS v6.1.3 and later use the `cpu_features` package by Google to determine if the target architecture supports AVX2 and/or AVX512F (on Intel x86_64 architectures only). In case you have build issues with this package, you can compile without it (and then AVX2 and AVX512F acceleration will not be used):

```
make CMAKE_OPTIONS='-DGBNCPUFEAT=1'
```

Without `cpu_features`, it is still possible to enable AVX2 and AVX512F. Rather than relying on run-time tests, you can use these flags to enable both AVX2 and AVX512F, without relying on `cpu_features`:

```
make CMAKE_OPTIONS='-DGBNCPUFEAT=1 -DGBAVX2=1 -DGBAVX512F=1'
```

To use multiple options, separate them by a space. For example, to build just the library but not `cpu_features`, and to enable AVX2 but not AVX512F, and use 40 threads to compile:

```
make CMAKE_OPTIONS='-DGBNCPUFEAT=1 -DGBAVX2=1' JOBS=40
```

After compiling the library, you can compile the demos with `make all` and then `make demos` while in the top-level GraphBLAS folder.

If `cmake` or `make` fail, it might be that your default compiler does not support C11. Try another compiler. For example, try one of these options. Go into the `build` directory and type one of these:

```
CC=gcc cmake ..
CC=gcc-11 cmake ..
CC=xlc cmake ..
CC=icx cmake ..
```

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

```
CC=gcc make
CC=gcc-11 make
CC=xlc make
CC=icx make
```

For faster compilation, you can specify a parallel make. For example, to use 32 parallel jobs and the `gcc` compiler, do the following:

```
JOBS=32 CC=gcc make
```

18.3.2 On the Mac (Intel or ARM)

GraphBLAS requires `cmake` v3.20 or later, and it optionally uses `make` to simplify the use of the cmake build system. It also needs other Apple Command Line Tools from Xcode. First install Xcode (see https://developer.apple.com/xcode), and then install the command line tools for Xcode:

```
xcode-select --install
```

Next, install brew, at https://brew.sh, or `spack`. Use brew or spack to install cmake version 3.20 or later.

18.3.3 On the Intel-based Mac

If you have the Intel compiler and its OpenMP library, then use the following in the top-level `GraphBLAS` folder. The Intel OpenMP library will be found automatically:

```
make CC=icc CXX=icpc
```

18.3.4 MATLAB on the Mac (Apple Silicon based)

MATLAB on the Apple-Silicon-based Mac is now a native ARM64 application (as of R2023b). GraphBLAS is not supported for earlier versions of MATLAB on Apple Silicon.

Note that GraphBLAS must use the same OpenMP library as MATLAB. This is handled by the `graphblas_install.m` script.

18.3.5 On Microsoft Windows

SuiteSparse:GraphBLAS can be compiled by the Microsoft C compiler (`cl`) using Microsoft Visual Studio. However, that compiler is not C11 compliant. As a result, GraphBLAS on Windows will have a few minor limitations.

- The MS Visual Studio compiler does not support the `_Generic` keyword, required for the polymorphic GraphBLAS functions. So for example, you will need to use `GrB_Matrix_free` instead of just `GrB_free`.
- Variable-length arrays are not supported, so user-defined types are limited to 1024 bytes in size. This can be changed by editing `GB_VLA_MAXSIZE` in `Source/GB_compiler.h`, and recompiling SuiteSparse:GraphBLAS.
- AVX acceleration is not enabled.
- You must compile with 64-bit computing enabled (x64). Otherwise, a compiler error will occur (`InterlockedExchange64`, `InterlockdOr64` and other methods will not be found).

If you use a recent `gcc` or `icx` compiler on Windows other than the Microsoft Compiler (`cl`), these limitations can be avoided.

The following instructions apply to Windows 10, CMake 3.16, and Visual Studio 2019, but may work for earlier versions.

1. Install CMake 3.16 or later, if not already installed. See https://cmake.org/ for details.
2. Install Microsoft Visual Studio, if not already installed. See https://visualstudio.microsoft.com/ for details. Version 2019 is preferred, but earlier versions may also work.
3. Open a terminal window and type this in the `GraphBLAS/build` folder:

   ```
   cmake ..
   ```

   Alternatively, use the `cmake` gui program to configure the cmake build system for GraphBLAS.
4. The `cmake` command generates many files in `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. Alternatively, instead of opening Visual Studio, type this command in the terminal window while in the `build` folder:

```
cmake --build . --config Release
```

8. Add the `GraphBLAS/build/Release` folder to the Windows System path:

   - Open the `Start Menu` and type `Control Panel`.
   - Select the `Control Panel` app.
   - When the app opens, select `System and Security`.
   - Under `System and Security`, select `System`.
   - From the top left side of the `System` window, select `Advanced System Settings`. You may have to authenticate at this step. If you cannot authenticate, try setting the User Environment Variables instead.
   - 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`. To use the MATLAB interface, add the full path of the `GraphBLAS\GraphBLAS\build\Release` folder as well.
- 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`.

9. 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/Octave interface discussed in the next section.

18.3.6 Mac using clang

To use OpenMP with clang on the Mac, try installing it from the https://mac.r-project.org/openmp/ project. Be sure to check that page for the OpenMP version that matches your version of Apple Xcode. For example, if using Xcode 13.3 to 13.4.1, use:

```
curl -O https://mac.r-project.org/openmp/openmp-13.0.0-darwin21-Release.tar.gz
sudo tar fvxz openmp-13.0.0-darwin21-Release.tar.gz -C /
```

These commands will install universal binaries (ARM and x86) for `libomp.dylib`, and the following files:

```
/usr/local/lib/libomp.dylib
/usr/local/include/ompt.h
/usr/local/include/omp.h
/usr/local/include/omp-tools.h
```

Once you do this, the GraphBLAS cmake build system should find this copy of OpenMP for clang.

18.3.7 Linking issues after installation

My Linux distro (Ubuntu 18.04) includes a copy of `libgraphblas.so.1`, which is SuiteSparse:GraphBLAS v1.1.2. After installing SuiteSparse:GraphBLAS in /usr/local/lib (with sudo make install), compiling a simple stand-alone program links against libgraphblas.so.1 instead of the latest version, while at the same time accessing the latest version of the include file as /usr/local/include/GraphBLAS.h. This command fails:

```
gcc prog.c -lgraphblas
```

Revising my LD_LIBRARY_PATH to put /usr/local/lib first in the library directory order didn't help. If you encounter this problem, try one of the following options (all four work for me, and link against the proper version, /usr/local/lib/libgraphblas.so.9.2.0 for example):

```
gcc prog.c -l:libgraphblas.so.9
gcc prog.c -l:libgraphblas.so.9.2.0
gcc prog.c /usr/local/lib/libgraphblas.so
gcc prog.c -Wl,-v -L/usr/local/lib -lgraphblas
```

This prog.c test program is a trivial one, which works in v1.0 and later:

```
#include <GraphBLAS.h>
int main (void)
{
    GrB_init (GrB_NONBLOCKING) ;
    GrB_finalize ( ) ;
}
```

Compile the program above, then use this command to ensure libgraphblas.so.9 appears:

```
ldd a.out
```

18.3.8 Running the tests

To run a short test, type make demo 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 R2018a or later 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.

18.3.9 Cleaning up

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

19 Release Notes

- Mar 1, 2025: version 10.0.0
  - 32/64 bit matrices and vectors: the `GrB_Matrix` and `GrB_Vector` now exploit 32-bit integers when possible. New methods added to pass 32-bit integer arrays to/from `GrB_build`, `extract`, `assign`, `subassign`, and `extractTuples`. New object, the `GxB_Container` added for fast import/export of matrices/vectors with arbitrary integer content.
  - `GrB_Field`: this enum is strongly deprecated, and replaced with `typedef int GrB_Field`. This is an upward-compatible change to the API, and will allow the creation of a future mathematical field object in GraphBLAS. This type should not be used; use an int instead. It will be replaced in a future version of GraphBLAS.
  - enum parameters: replaced all enum parameters with int, to simplify future updates to enum parameters, including the `GrB_Field`.
  - `GxB_JIT_ERROR`: added in 9.4.x, changed value to avoid conflict with LAGraph error codes.
  - AppleClang compiler bug: On the Mac, the `Source/mask/GB_masker.c` file triggers a bug in AppleClang 16.0.0 with -O3 in (MacOS 14.6.1 (23G93), Xcode 16.2, Apple clang version 16.0.0, clang-1600.0.26.6). It also fails in MacOS 15.2 (Target: arm64-apple-darwin23.6.0). The bug is triggered by these tests in LAGraph (v1.2 branch, unreleased, Jan 4, 2025):

```
39 - LAGraphX_BF (SIGTRAP)
40 - LAGraphX_Coarsen_Matching (Failed)
41 - LAGraphX_FastGraphletTransform (SIGTRAP)
49 - LAGraphX_PageRankGX (SIGTRAP)
54 - LAGraphX_SquareClustering (SIGTRAP)
61 - LAGraphX_msf (Failed)
```

When using clang, optimization is turned off for this file. This has no impact on performance since `GB_masker.c` is very simple, consisting of a single sequence of calls to other methods.

  - pack/unpack: these are declared historical; they still work but use 64-bit integers only. Use the new `GxB_Container` methods instead.
  - `GxB_Matrix_iso` and `GxB_Vector_iso`: declared historical; use `GrB_get` with the new `GxB_ISO` enum.
  - `GxB_Matrix_type`, `GxB_Vector_type`, `GxB_Scalar_type`: no longer historical; added back to the user guide.
  - Summary: the API is upward-compatible with 9.4.x, but only after the user application is recompiled with GraphBLAS v10.0.0. As a result, the SO version must increase from 9 to 10.

- Dec 20, 2024: version 9.4.3

  - (57) bug fix: GraphBLAS.h header: remove duplicate definitions of `GxB_MAX_FIRST_*` semirings (incompletely moved to 'historical' section in 9.4.2).

- Nov 20, 2024: version 9.4.2

  - clarified User Guide: regarding when the hyper-hash is built
  - JIT: reduced JIT kernel encodings
  - (also includes the updates from 9.4.0.beta and 9.4.1.beta listed below).

- Nov 15, 2024: version 9.4.1 (only released as BETA)

  - More JIT kernels: all JIT kernels for `GrB_assign`, `GxB_subassign`, `GrB_extract`, `GxB_sort GrB_kronecker`, the stand-alone mask phase (an internal method that computes $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$), and utilities have been created. All kernels formerly tagged in the code as `JIT: needed` are now finished.

- removed Factory kernels for: types int8 and uint8, and semirings: max_min, max_plus, max_times, min_max, min_times, plus_min, plus_max, non-Boolean land/lor/lxor/lxnor, and integer times_first/second, to reduce size of compiled library. JIT kernels will be used instead for these types and semirings.
- `GxB_IndexBinaryOp`: finalized and named as `GxB_*`.

- Oct 15, 2024: version 9.4.0 (only released as BETA)
  - new operator and associated methods: added the draft `G*B_IndexBinaryOp`.
  - JIT error-handling behavior changed: if a compiler error occurs in the JIT, `GxB_JIT_ERROR` is now returned. Previously, GraphBLAS would fall back to a generic method if such an error occurred.

- Aug 12, 2024: version 9.3.1
  - (56) bug fix: wrong type for fgetc return value in JITpackage; leads to infinite loop on some systems when building GraphBLAS.

- Aug 2, 2024: version 9.3.0
  - code restructuring: Source folder split into many subfolders, and some files and internal functions renamed. No visible external change.
  - (55) bug fix: `GrB_apply` with user-defined index-unary op and generic kernel.
  - (54) bug fix: reducing a huge iso full matrix to a scalar resulted in integer overflow if nrows*ncols was larger than about 2^{60}.
  - reduced size of compiled library: `int16` and `uint16` types and operators for FactoryKernels are disabled in `GB_control.h`. The JIT will always be used instead.

- May 22, 2024: version 9.2.0
  - Added `graphblas_install.m` for a simpler method of compiling the MATLAB/Octave interface for GraphBLAS.

  - JIT: sanitizing the JIT cache path, better burble for compiler errors.
  - `GrB_get/GrB_set`: better handling of concurrent get/set between different user threads.

- Mar 22, 2024: version 9.1.0

  - minor updates to build system
  - C11 complex type detection: this is now detected and configured by cmake, instead of using an `#if` ... in the GraphBLAS.h header. This change was required to port GraphBLAS to the clang-cl compiler on Windows when it simulates the MSVC compiler. Also added a new feature (thus the minor version update to 9.1.0): `GxB_HAVE_COMPLEX*` to GraphBLAS.h to indicate which kind of complex data types are available in C11 or MSVC. Contributed by Markus Mützel.
  - (53) bug fix: eWiseAdd C<M>=A+B when M, A, and B are all hypersparse; access to M was incorrect (also affects C<M>+=T for any operation, if M and T are both hypersparse).

- Mar 1, 2024: version 9.0.3

  - (52) performance bug fix: JIT kernels since v8.3.1 were not compiled with OpenMP.

- Feb 26, 2024: version 9.0.2

  - GraphBLAS/Makefile `make static` was incorrect.

- Jan 20, 2024: version 9.0.1

  - minor updates to build system

- Version 9.0.0, Jan 10, 2024

  - `GrB_get/GrB_set`: new functions from the v2.1 C API.

- GrB_Type_new, GrB_UnaryOp_new, GrB_IndexUnaryOp_new: no longer macros, since GrB_set can be used to set the names of the operators. These methods no longer extract the name, so the default name is now the empty string. This is because GrB_get/set can only set these names once. This is a non-compatible change of behavior for these 3 methods, so SuiteSparse:GraphBLAS must become v9.0.0.
- historical methods: many methods are replaced by GrB_get and GrB_set. They remain in SuiteSparse:GraphBLAS but have been declared historical. Terse prototypes exist in GraphBLAS.h, and any discussion is removed from the User Guide: GxB_get, GxB_set, and the methods they call, and many more. Use GrB_get/set in place those methods, and for: GxB_*type_name, GxB_*type, GxB_Monoid_operator, GxB_Monoid_identity, GxB_Monoid_terminal, GxB_Semiring_add, GxB_Semiring_multiply. Use GrB_STORAGE_ORIENTATION_HINT in place of GxB_FORMAT.
- hyper_hash: constructed only if the number of non-empty vectors in a hypersparse matrix is large (> 1024, by default).
- minor updates to build system: *.pc files for pkgconfig

- Dec 30, 2023: version 8.3.1
  - remove #undef I from GraphBLAS.h, so as not to conflict with the definition of I from complex.h.
  - major change to build system: by Markus Mützel

- Oct 7, 2023: version 8.2.1
  - (49) bug fix: GrB_mxm saxpy4 and saxpy5 had incorrectly handling of typecasting in v8.0.0 to v8.2.0 (caught by Erik Welch)
  - cross-compiler support: replace check_c_source_runs with _compiles for GraphBLAS and SuiteSparse_config, and remove check for getenv("HOME").
  - cmake update: add "None" build type, from Antonio Rojas, for Arch Linux

- Version 8.2.0, Sept 8, 2023

– cmake updates: `SuiteSparse::` namespace by Markus Mützel.

- Version 8.0.2, June 16, 2023

  – added `-DJITINIT=option`: use `-DJITINIT` to set the initial state of the `GxB_JIT_C_CONTROL` (4:on, 3:load, 2:run, 1:pause, 0:off). The default is 4 (on) if the JIT is enabled, or 2 (run) if `-DNJIT=1` is set.

  – xxHash: upgraded to latest version as of June 16, 2023

- Version 8.0.1, May 27, 2023

  – (48) bug fix: `GrB_*_nvals` returned `UINT64_MAX` ('infinity') for a `GrB_Vector` of size n-by-2^{60}; it should return 2^{60}. Caught by Erik Welch, NVIDIA.

  – added `GxB_Context_error` and `GxB_Context_wait`

  – `C++`: changed complex typedefs for `C++` that include GraphBLAS.h. Update from Markus Mützel.

- Version 8.0.0 (May 18, 2023)

  – version 8: This version is a major SO version increase, since it removes a few minor user-visible features from SuiteSparse:GraphBLAS: the `GrB_Descriptor` no longer supports threading control, and some features of the `GxB_SelectOp` are removed (see below). Enum values have been changed for compatibility with the upcoming `GrB_set/get` features in the V2.1 C API.

  – The JIT: GraphBLAS v8.0.0 includes a JIT for the CPU kernels, which can compile kernels at run-time. Added `GxB_set/get` options and environment variables to control the JIT. The `GxB_*Op_new` methods can accept NULL function pointers, if the strings are provided, valid, and the JIT is enabled.

  – `GxB_Type_new`: the size of the type can be given as zero, in which case the size is determined via a JIT kernel.

  – `GxB_UnaryOp_new`, `GxB_BinaryOp_new`, and `GxB_IndexUnaryOp_new`: the function pointer can be given as `NULL`, in which case the function is created by the JIT.

- math kernels: revised for CUDA JIT. More accurate complex floating-point for Mac OS on Apple Silicon.
- Demo/wildtype_demo: change to double so that CPU and GPU versions compute the same result.
- GxB_get: can return malloc/calloc/realloc/free
- GxB_Context: an object for controlling computational resources: number of OpenMP threads, the chunk factor, and (draft) GPU id.
- GrB_Descriptor: removed ability to control the number of OpenMP threads from the descriptor (a rarely used feature). Replaced with the GxB_Context object.
- GxB_SelectOp: GraphBLAS no longer supports user-defined GxB_SelectOps. Use a GrB_IndexUnaryOp instead. The GxB_SelectOp_new and GxB_SelectOp_free functions are removed entirely. The built-in GxB_SelectOps, GxB_Matrix_select, GxB_Vector_select, and GxB_select still work. However, the GxB_EQ_THUNK, GxB_EQ_ZERO, GxB_NE_THUNK, and GxB_NE_ZERO operators no longer work on user-defined types, as they did in v7.4.4 and earlier. Create a user-defined GrB_IndexUnaryOp to compute these operations instead, and use GrB_select.
- alternative/Makefile: removed; not compatible with the JIT.
- zstd: upgraded to v1.5.5 (Apr 4, 2023)

- Version 7.4.4 (Mar 25, 2023)
  - (47) bug fix: OpenMP atomics require seq_cst on the ARM. Revised GB_atomics.h accordingly, and added them for all architectures (caught by Gábor Szárnyas).

- Version 7.4.3 (Jan 20, 2023)
  - debug: turned on in GrB_Matrix_removeElement by mistake.

- Version 7.4.2 (Jan 17, 2023)
  - minor change to build system: for SuiteSparse v7.0.0.

  - deprecation notice: in GraphBLAS v8.0.0, the ability to set the number of threads, and chunk size, in the descriptor will be removed. It still appears in v7.x, but will be replaced by a Context object in v8.0.0.

- Version 7.4.1 (Dec 29, 2022)

  - global free pool: disabled. Benefit for single-thread user applications was modest, and it causes too much contention in a critical section when the user application is multi-threaded.
  - `GrB_mxm`: revised task creation heuristics for sparse-times-sparse for better performance.

- Version 7.4.0 (Dec 23, 2022)

  - add non-`va_arg` methods: `va_arg`-based `GxB_get/set` methods are C11 but cause issues for cffi in Python. As a temporary workaround, new methods have been added that do not use `va_arg`. The existing `GxB_get/set` methods are not changed. The new methods are not in the user guide, since all of the `GxB_get/set` methods will be superceded with `GrB_get/set` in the v2.1 C API. At that point, all `GxB_get/set` methods will become historical (kept, not deprecated, but removed from the user guide).

- Version 7.3.3 (Dec 9, 2022)

  - `stdatomic.h`: using `#include <stdatomic.h>` and `atomic_compare_exchange_weak` instead of GCC/clang/icx `__atomic_*` variants. Added `-latomic` if required.
  - chunk factor for C=A*B (saxpy3 method): revised for non-builtin-semirings

- Version 7.3.2 (Nov 12, 2022)

  - `cmake_modules`: minor revision to build system, to sync with SuiteSparse v6.0.0
  - Added option `-DNOPENMP=1` to disable OpenMP parallelism.

- Version 7.3.1 (Oct 21, 2022)
  - workaround for a bug in the Microsoft Visual Studio Compiler, MSC 19.2x (in vs2019).

- Version 7.3.0 (Oct 14, 2022)
  - `GrB_Matrix`: changes to the internal data structure
  - minor internal changes: `A->nvals` for sparse/hypersparse
  - more significant changes: added hyper-hash for hypersparse case, speeds up many operations on hypersparse matrices. Based on [Gre21].
  - `GxB_unpack_HyperHash` and `GxB_pack_HyperHash`: to pack/unpack the hyper-hash
  - `@GrB` MATLAB/Octave interface: changed license to Apache-2.0.
  - MATLAB library: renamed to `libgraphblas_matlab.so`
  - performance: faster `C=A*B` when using a single thread and `B` is a sparse vector with many entries.

- Version 7.2.0 (Aug 8, 2022)
  - added ZSTD as a compression option for serialize/deserialize: Version 1.5.3 by Yann Collet, https://github.com/facebook/zstd.git. Copyright (c) 2016-present, Facebook, Inc. All rights reserved. Included in SuiteSparse:GraphBLAS via its BSD-3-clause license. The default method is now ZSTD, level 1.
  - `GxB_Matrix_reshape*` added.
  - MATLAB interface: `reshape`, `C(:)=A`, `C=A(:)` are faster. Better error messages.

- Version 7.1.2 (July 8, 2022)
  - MATLAB interface: linear indexing added for C(:)=A, C=A(:), and single-output I=find(C). Faster bandwidth, istriu, istril, isbanded, isdiag. C(I,J)=A can now grow the size of A.

- Version 7.1.1 (June 3, 2022)
  - minor updates to documentation and error messages

- MATLAB interface: minor revision of GrB.deserialize

- Version 7.1.0 (May 20, 2022)
  - added cube root: `GxB_CBRT_FP32` and `GxB_CBRT_FP64` unary operators
  - added `GxB_Matrix_isStoredElement` and `GxB_Vector_isStoredElement`

- Version 7.0.4 (Apr 25, 2022)
  - (46) bug fix: user-defined type size was incorrectly limited to 128 bytes. Caught by Erik Welch.

- Version 7.0.3 (Apr 8, 2022)
  - faster transpose when using 2 threads

- Version 7.0.2 (Apr 5, 2022)
  - (45) bug fix: vector iterator was broken for iterating across a vector in bitmap format. Caught by Erik Welch.

- Version 7.0.1 (Apr 3, 2022)
  - added revised ACM TOMS submission to the Doc folder

- Version 7.0.0 (Apr 2, 2022)
  - (44) spec bug: `GrB_Matrix_diag` was implemented in v5.2.x and v6.x with the wrong signature. This fix requires the major release to change, from v6.x to v7.x.
  - (43) performance bug fix for `GrB_mxm`: auto selection for saxpy method (Hash vs Gustavson) revised.
  - `GrB_assign`: better performance for `C(i,j)=scalar` and `C(i,j)+=scalar` when `i` and `j` have length 1 (scalar assigment with no scalar expansion).

- Version 6.2.5 (Mar 14, 2022)
  - For SuiteSparse v5.11.0.

- Version 6.2.4 (Mar 8, 2022)
  - (42) bug fix: `GrB_mxm` with 0-by-0 iso full matrices. Caught by Henry Amuasi in the Python grblas interface, then triaged and isolated by Erik Welch.
- Version 6.2.3 (Mar 5, 2022)
  - minor update to documentation in `GrB.build`: no change to any code
- Version 6.2.2 (Feb 28, 2022)
  - revised output of `GxB_*_sort` to return newly created matrices C and P as full or bitmap matrices, as appropriate, instead of sparse/hypersparse, following their sparsity control settings.
- Version 6.2.1 (Feb 14, 2022)
  - (41) bug fix: `GxB_Iterator_get` used `(void *) + size` arithmetic
- Version 6.2.0 (Feb 14, 2022)
  - added the `GxB_Iterator` object and its methods. See Section 14.
  - `@GrB` interface: revised sparse-times-full rule for the conventional semiring (the syntax `C=A*B`), so that sparse-times-full results in `C` as full, but hypersparse-times-sparse is not full (typically sparse or hypersparse).
- Version 6.1.4 (Jan 12, 2022)
  - added Section 16 to User Guide: how to get the best performance out of algorithms based on GraphBLAS.
  - `cpu_features`: no longer built as a separate library, but built directly into `libgraphblas.so` and `libgraphblas.a`. Added compile-time flags to optionally disable the use of `cpu_features` completely.
  - Octave 7: port to Apple Silicon (thanks to Gábor Szárnyas).

- min/max monoids: real case (FP32 and FP64) no longer terminal
- @GrB interface: overloaded C=A*B syntax where one matrix is full always results in a full matrix C.
- Faster C=A*B for sparse-times-full and full-times-sparse for @GrB interface.

- Version 6.1.3 (Jan 1, 2022)
  - performance: task creation for GrB_mxm had a minor flaw (results were correct but parallelism suffered). Performance improvement of up to 10x when nnz(A)¡¡nnz(B).

- Version 6.1.2 (Dec 31, 2021)
  - performance: revised swap_rule in GrB_mxm, which decides whether to compute C=A*B or C=(B'*A')', and variants, resulting in up to 3x performance gain over v6.1.1 for GrB_mxm (observed; could be higher in other cases).

- Version 6.1.1 (Dec 28, 2021)
  - minor revision to AVX2 and AVX512f selection
  - cpu_features/Makefile: remove test of list_cpu_features so that the package can be built when cross-compiling

- Versions 6.1.0 (Dec 26, 2021)
  - added GxB_get options: compiler name and version.
  - added package: https://github.com/google/cpu_features, Nov 30, 2021 version.
  - performance: faster C+=A*B when C is full, A is bitmap/full, and B is sparse/hyper. Faster C+=A'*B when A is sparse/hyper, and B is bitmap/full.
  - (40) bug fix: deserialization of iso and empty matrices/vectors was incorrect

- Versions 6.0.2 and 5.2.2 (Nov 30, 2021)

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

- Versions 6.0.1 and 5.2.1 (Nov 27, 2021)

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

- Version 6.0.0 (Nov 15, 2021)

  - this release contains only a few changes that cause a break with backward compatibility. It is otherwise identical to v5.2.0.
  - v6.0.0 is fully compliant with the v2.0 C API Specification. Three changes from the v2.0 C API Spec are not backward compatible (GrB_*wait, GrB_Info, GrB_SCMP). GxB_init has also changed.
    - GrB_wait (object, mode): was GrB_wait (&object).
    - GrB_Info: changed enum values
    - GrB_SCMP: removed
    - GxB_init (mode, malloc, calloc, realloc, free, is_thread_safe): the last parameter, is_thread_safe, is deleted. The malloc, calloc, realloc, and free functions must be thread-safe.

- Version 5.2.0 (Nov 15, 2021)

  - Added for the v2.0 C API Specification: only features that are backward compatible with SuiteSparse:GraphBLAS v5.x have been added to v5.2.0:
    - GrB_Scalar: replaces GxB_Scalar, GxB_Scalar_* functions renamed GrB
    - GrB_IndexUnaryOp: new, free, fprint, wait
    - GrB_select: selection via GrB_IndexUnaryOp
    - GrB_apply: with GrB_IndexUnaryOp

    * GrB_reduce: reduce matrix or vector to GrB_Scalar
    * GrB_assign, GrB_subassion: with GrB_Scalar input
    * GrB_*_extractElement_Scalar: get GrB_Scalar from a matrix or vector
    * GrB*build: when dup is NULL, duplicates result in an error.
    * GrB import/export: import/export from/to user-provided arrays
    * GrB_EMPTY_OBJECT, GrB_NOT_IMPLEMENTED: error codes added
    * GrB_*_setElement_Scalar: set an entry in a matrix or vector, from a GrB_Scalar
    * GrB_Matrix_diag: same as GxB_Matrix_diag (C,v,k,NULL)
    * GrB_*_serialize/deserialize: with compression
    * GrB_ONEB_T: binary operator, $f(x, y) = 1$, the same as GxB_PAIR_T.
  - GxB*import* and GxB*export*: now historical.
  - GxB_select: is now historical; use GrB_select instead.
  - GxB_IGNORE_DUP: special operator for build methods only; if dup is this operator, then duplicates are ignored (not an error)
  - GxB_IndexUnaryOp_new: create a named index-unary operator
  - GxB_BinaryOp_new: create a named binary operator
  - GxB_UnaryOp_new: create a named unary operator
  - GxB_Type_new: to create a named type
  - GxB_Type_name: to query the name of a type
  - added GxB_*type_name methods to query the name of a type as a string.
  - GxB_Matrix_serialize/deserialize: with compression; optional descriptor.
  - GxB_Matrix_sort, GxB_Vector_sort: sort a matrix or vector
  - GxB_eWiseUnion: like GrB_eWiseAdd except for how entries in $\mathbf{A} \setminus \mathbf{B}$ and $\mathbf{B} \setminus \mathbf{A}$ are computed.
  - added LZ4/LZ4HC: compression library, http://www.lz4.org (BSD 2), v1.9.3, Copyright (c) 2011-2016, Yann Collet.
  - MIS and pagerank demos: removed; MIS added to LAGraph/experimental
  - disabled free memory pool if OpenMP not available
  - (37) bug fix: ewise C=A+B when all matrices are full, GBCOMPACT not used, but GB_control.h disabled the operator or type. Caught by Roi Lipman, Redis.

  - (36) bug fix: `C<M>=Z` not returning `C` as iso if `Z` iso and `C` initially empty. Caught by Erik Welch, Anaconda.
  - performance improvements: `C=A*B`: sparse/hyper times bitmap/full, and visa versa, including `C += A*B` when `C` is full.

- Version 5.1.10 (Oct 27, 2021)

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

- Version 5.1.9 (Oct 26, 2021)

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

- Version 5.1.8 (Oct 5, 2021)

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

- Version 5.1.7 (Aug 23, 2021)

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

- Version 5.1.6 (Aug 16, 2021)

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

- Version 5.1.5 (July 15, 2021)

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

- Version 5.1.4 (July 6, 2021)

  - faster Octave interface. Octave v7 or later is required.

- (30) bug fix: 1-based printing not enabled for pending tuples. Caught by Will Kimmerer, while working on the Julia interface.

- Version 5.1.3 (July 3, 2021)
  - added `GxB_Matrix_iso` and `GxB_Vector_iso`: to query if a matrix or vector is held as iso-valued
  - (29) bug fix: `Matrix_pack_*R` into a matrix previously held by column, or `Matrix_pack*C` into a matrix by row, would flip the dimensions. Caught by Erik Welch, Anaconda.
  - (28) bug fix: `kron(A,B)` with iso input matrices `A` and `B` fixed. Caught by Michel Pelletier, Graphegon.
  - (27) bug fix: v5.1.0 had a wrong version of a file; posted by mistake. Caught by Michel Pelletier, Graphegon.

- Version 5.1.2 (June 30, 2021)
  - iso matrices added: these are matrices and vectors whose values in the sparsity pattern are all the same. This is an internal change to the opaque data structures of the `GrB_Matrix` and `GrB_Vector` with very little change to the API.
  - added `GxB_Matrix_build_Scalar` and `GxB_Vector_build_Scalar`, which always build iso matrices and vectors.
  - import/export methods can now import/export iso matrices and vectors.
  - added `GrB.argmin/argmax` to MATLAB/Octave interface
  - added `GxB_*_pack/unpack` methods in place of import/export.
  - added `GxB_PRINT_1BASED` to the global settings.
  - added `GxB_*_memoryUsage`
  - port to Octave: `gbmake` and `gbtest` work in Octave7 to build and test the `@GrB` interface to GraphBLAS. Octave 7.0.0 is required.

- Version 5.0.6 (May 24, 2021)
  - BFS and triangle counting demos removed from GraphBLAS/Demo: see LAGraph for these algorithms. Eventually, all of GraphBLAS/Demo will be deleted, once LAGraph includes all the methods included there.

- Version 5.0.5 (May 17, 2021)
  - (26) performance bug fix: reduce-to-vector where A is hypersparse CSR with a transposed descriptor (or CSC with no transpose), and some cases for GrB_mxm/mxv/vxm when computing C=A*B with A hypersparse CSC and B bitmap/full (or A bitmap/full and B hypersparse CSR), the wrong internal method was being selected via the auto-selection strategy, resulting in a significant slowdown in some cases.
- Version 5.0.4 (May 13, 2021)
  - @GrB MATLAB/Octave interface: changed license to GNU General Public License v3.0 or later. It was licensed under Apache-2.0 in Version 5.0.3 and earlier. Changed back to Apache-2.0 for Version 7.3.0; see above.
- Version 5.0.3 (May 12, 2021)
  - (25) bug fix: disabling ANY_PAIR semirings by editing Source/GB_control.h would cause a segfault if those disabled semirings were used.
  - demos are no longer built by default
  - (24) bug fix: new functions in v5.0.2 not declared as extern in GraphBLAS.h.
  - GrB_Matrix_reduce_BinaryOp reinstated from v4.0.3; same limit on built-in ops that correspond to known monoids.
- Version 5.0.2 (May 5, 2021)
  - (23) bug fix: GrB_Matrix_apply_BinaryOp1st and 2nd were using the wrong descriptors for GrB_INP0 and GrB_INP1. Caught by Erik Welch, Anaconda.
  - memory pool added for faster allocation/free of small blocks
  - @GrB interface ported to MATLAB R2021a.
  - GxB_PRINTF and GxB_FLUSH global options added.
  - GxB_Matrix_diag: construct a diagonal matrix from a vector
  - GxB_Vector_diag: extract a diagonal from a matrix
  - concat/split: methods to concatenate and split matrices.

- import/export: size of arrays now in bytes, not entries. This change is required for better internal memory management, and it is not backward compatible with the GxB*import/export functions in v4.0. A new parameter, is_uniform, has been added to all import/export methods, which indicates that the matrix values are all the same.
- (22) bug fix: SIMD vectorization was missing reduction(+,task_cnvals) in GB_dense_subassign_06d_template.c. Caught by Jeff Huang, Texas A&M, with his software package for race-condition detection.
- GrB_Matrix_reduce_BinaryOp: removed. Use a monoid instead, with GrB_reduce or GrB_Matrix_reduce_Monoid.

- Version 4.0.3 (Jan 19, 2021)
  - faster min/max monoids
  - G=GrB(G) converts G from v3 object to v4

- Version 4.0.2 (Jan 13, 2021)
  - ability to load *.mat files saved with the v3 GrB

- Version 4.0.1 (Jan 4, 2021)
  - significant performance improvements: compared with v3.3.3, up to 5x faster in breadth-first-search (using LAGraph_bfs_parent2), and 2x faster in Betweenness-Centrality (using LAGraph_bc_batch5).
  - GrB_wait(void), with no inputs: removed
  - GrB_wait(&object): polymorphic function added
  - GrB_*_nvals: no longer guarantees completion; use GrB_wait(&object) or non-polymorphic GrB_*_wait (&object) instead
  - GrB_error: now has two parameters: a string (char **) and an object.
  - GrB_Matrix_reduce_BinaryOp limited to built-in operators that correspond to known monoids.
  - GrB_*_extractTuples: may return indices out of order
  - removed internal features: GBI iterator, slice and hyperslice matrices

  - bitmap/full matrices and vectors added
  - index-based operators and semirings: GxB_FIRSTI_INT32 and related ops
  - jumbled matrices: sort left pending, like zombies and pending tuples
  - GxB_get/set: added GxB_SPARSITY_* (hyper, sparse, bitmap, or full) and GxB_BITMAP_SWITCH.
  - GxB_HYPER: enum renamed to GxB_HYPER_SWITCH
  - GxB*import/export: API modified
  - GxB_SelectOp: nrows and ncols removed from function signature.
  - OpenMP tasking removed from mergesort and replaced with parallel for loops. Just as fast on Linux/Mac; now the performance ports to Windows.
  - GxB_BURBLE added as a supported feature. This was an undocumented feature of prior versions.
  - bug fix: A({lo,hi})=scalar A(lo:hi)=scalar was OK

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

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

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

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

  Added complex types (GxB_FC32 and GxB_FC64), many unary operators, binary operators, monoids, and semirings. Added bitwise operators, and their monoids and semirings. Added the predefined monoids and semirings from the v1.3 specification. @GrB interface: added complex matrices and operators, and changed behavior of integer operations to more closely match the behavior on built-in integer matrices. The rules for typecasting large floating point values to integers has changed. The specific object-based GrB_Matrix_wait, GrB_Vector_wait, etc,

functions have been added. The no-argument `GrB_wait()` is deprecated. Added `GrB_getVersion`, `GrB_Matrix_resize`, `GrB_Vector_resize`, `GrB_kronecker`, `GrB_*_wait`, scalar binding with binary operators for `GrB_apply`,
`GrB_Matrix_removeElement`, and `GrB_Vector_removeElement`.

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

- Version 3.1.2 (Dec, 2019). Changes to allow SuiteSparse:GraphBLAS to be compiled with the Microsoft Visual Studio compiler. This compiler does not support the `_Generic` keyword, so the polymorphic functions are not available. Use the equivalent non-polymorphic functions instead, when compiling GraphBLAS with MS Visual Studio. In addition, variable-length arrays are not supported, so user-defined types are limited to 128 bytes in size. These changes have no effect if you have an C11 compliant compiler.

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

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

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

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

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

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

- Version 2.3 (Feb 2019) improves the performance of many GraphBLAS operations, including an early-exit for monoids. These changes have a significant impact on breadth-first-search (a performance bug was also fixed in the two BFS `Demo` codes). The matrix and vector import/export functions were added (Section **??**), in support of the new LAGraph project (https://github.com/GraphBLAS/LAGraph, see also Section 17.1). 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 13). PageRank demos have been added to the `Demos` folder.

- Version 2.1 (Oct 2018) was a major update with support for new matrix formats (by row or column, and hypersparse matrices), and colon notation (`I=begin:end` or `I=begin:inc:end`). Some graph algorithms are more naturally expressed with matrices stored by row, and this version includes the new `GxB_BY_ROW` format. The default format in Version 2.1 and prior versions is by column. New extensions to GraphBLAS in this version include `GxB_get`, `GxB_set`, and `GxB_AxB_METHOD`, `GxB_RANGE`, `GxB_STRIDE`, and `GxB_BACKWARDS`, and their related definitions, described in Sections 6.14, 10, and 11.

- 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.

19.1 Regarding historical and deprecated functions and symbols

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

The following methods have been fully deprecated and removed. The older versions of GrB_wait and GrB_error methods have been removed since they are incompatible with the latest versions, per the C API Specification. The GxB_SelectOp_new and GxB_SelectOp_free methods have been removed, and some of the built-in operators have been been revised (specifically, the GxB_EQ_THUNK, GxB_EQ_ZERO, GxB_NE_THUNK, and GxB_NE_ZERO operators no longer work on user-defined types).

20 Acknowledgments

I would like to thank Jeremy Kepner (MIT Lincoln Laboratory Supercomputing Center), and the GraphBLAS API Committee: Aydın Buluç (Lawrence Berkeley National Laboratory), Timothy G. Mattson (Intel Corporation) Scott McMillan (Software Engineering Institute at Carnegie Mellon University), José Moreira (IBM Corporation), Carl Yang (UC Davis), and Benjamin Brock (UC Berkeley), for creating the GraphBLAS specification and for patiently answering my many questions while I was implementing it.

I would like to thank Tim Mattson and Henry Gabb, Intel, Inc., for their collaboration and for the support of Intel.

I would like to thank Joe Eaton and Corey Nolet for their collaboration on the CUDA kernels (still in progress), and for the support of NVIDIA.

I would like to thank Pat Quillen for his collaboration and for the support of MathWorks.

I would like to thank John Eaton for his collaboration on the integration with Octave 7.

I would like to thank Michel Pelletier for his collaboration and work on the pygraphblas interface, and Jim Kitchen and Erik Welch for their work on Anaconda's python interface.

I would like to thank Will Kimmerer for his collaboration and work on the Julia interface.

I would like to thank John Gilbert (UC Santa Barbara) for our many discussions on GraphBLAS, and for our decades-long conversation and collaboration on sparse matrix computations.

I would like to thank Sébastien Villemot (Debian Developer, http://sebastien.villemot.name) for helping me with various build issues and other code issues with GraphBLAS (and all of SuiteSparse) for its packaging in Debian Linux.

I would like to thank Gábor Szárnyas for porting the `@GrB` interface to Octave 7 on Apple Silicon.

I would like to thank Markus Mützel for his help in porting the updates to GraphBLAS v7.4.1 and SuiteSparse v7.0.0 to Windows.

I would like to thank Roi Lipman, Redis (https://redislabs.com), for our many discussions on GraphBLAS and for enabling its use in RedisGraph (https://redislabs.com/redis-enterprise/technology/redisgraph/), a graph database module for Redis. Based on SuiteSparse:GraphBLAS, RedisGraph is up 600x faster than the fastest graph databases (https://youtu.be/9h3Qco_x0QE https://redislabs.com/blog/new-redisgraph-1-0-achieves-600x-faster-performance-graph-databases/).

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

21 Additional Resources

See http://graphblas.org for the GraphBLAS community page. See https://github.com/GraphBLAS/GraphBLAS-Pointers for an up-to-date list of additional resources on GraphBLAS, maintained by Gábor Szárnyas.

References

[ACD+20] Mohsen Aznaveh, Jinhao Chen, Timothy A. Davis, Bálint Hegyi, Scott P. Kolodziej, Timothy G. Mattson, and Gábor Szárnyas. Parallel GraphBLAS with OpenMP. In *CSC20, SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 2020. https://www.siam.org/conferences/cm/conference/csc20.

[BBM+21] B. Brock, A. Buluç, T. Mattson, S. McMillan, and J. Moreira. The GraphBLAS C API specification (v2.0). Technical report, GraphBLAS.org, 2021. http://graphblas.org/.

[BG08] A. Buluç and J. Gilbert. On the representation and multiplication of hypersparse matrices. In *IPDPS'80: 2008 IEEE Intl. Symp. on Parallel and Distributed Processing*, pages 1–11, April 2008. https://dx.doi.org/10.1109/IPDPS.2008.4536313.

[BG12] A. Buluç and J. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012. https://dx.doi.org/10.1137/110848244.

[BMM+17a] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652, May 2017. https://dx.doi.org/10.1109/IPDPSW.2017.117.

[BMM+17b] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. The GraphBLAS C API specification. Technical report, GraphBLAS.org, 2017. http://graphblas.org/.

[DAK19] T. A. Davis, M. Aznaveh, and S. Kolodziej. Write quick, run fast: Sparse deep neural network in 20 minutes of development time via SuiteSparse:GraphBLAS. In *IEEE HPEC'19*. IEEE, 2019. Grand Challenge Champion, for high performance. See http://www.ieee-hpec.org/.

[Dav06] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, 2006. https://dx.doi.org/10.1137/1.9780898718881.

[Dav18] T. A. Davis. Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and K-truss. In *IEEE HPEC'18*. IEEE, 2018. Grand Challenge Innovation Award. See http://www.ieee-hpec.org/.

[Dav19] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), December 2019. https://doi.org/10.1145/3322125.

[Dav23] Timothy A. Davis. Algorithm 1037: Suitesparse:graphblas: Parallel graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 49(3), sep 2023.

[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. https://dx.doi.org/10.1017/S0962492916000076.

[Gre21] Oded Green. HashGraph – scalable hash tables using a sparse graph data structure. *ACM Trans. Parallel Comput.*, 8(2), July 2021. https://doi.org/10.1145/3460872.

[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, MIT, 2017. http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf.

[KG11] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra.* SIAM, Philadelphia, PA, 2011.

> From the preface: Graphs are among the most important abstract data types in computer science, and the algorithms that operate on them are critical to modern life. Graphs have been shown to be powerful tools for modeling complex problems because of their simplicity and generality. Graph algorithms are one of the pillars of mathematics, informing research in such diverse areas as combinatorial optimization, complexity theory, and topology. Algorithms on graphs are applied in many ways in today's world – from Web rankings to metabolic networks, from finite element meshes to semantic graphs. The current exponential growth in graph data has forced a shift to parallel computing for executing graph algorithms. Implementing parallel graph algorithms and achieving good parallel performance have proven difficult. This book addresses these challenges by exploiting the well-known duality between a canonical representation of graphs as abstract collections of vertices and edges and a sparse adjacency matrix representation. This linear algebraic approach is widely accessible to scientists and engineers who may not be formally trained in computer science. The authors show how to leverage existing parallel matrix computation techniques and the large amount of software infrastructure that exists for these computations to implement efficient and scalable parallel graph algorithms. The benefits of this approach are reduced algorithmic complexity, ease of implementation, and improved performance. DOI: https://dx.doi.org/10.1137/1.9780898719918

[MBM+24] T. G. Mattson, M. Bezbaruah, M. Maier, S. McMillan, M. Peletier, E. Welch, and T. A. Davis. Indexed binary operations in the graphblas. In *IEEE HPEC'24*. IEEE, 2024.

[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. https://doi.org/10.1145/3229710.3229720.

[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.