User Guide for SuiteSparse:GraphBLAS

Timothy A. Davis
davis@tamu.edu, Texas A&M University.
http://www.suitesparse.com and http://aldenmath.com

VERSION 2.1.0, Sept 11, 2018 (BETA1)

Abstract

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

NOTE: Version 2.1 is a major update with support for new matrix formats (by row or column, and hypersparse matrices), and MATLAB-like colon notation (I=begin:end or I=begin:inc:end). Some graph algorithms are more naturally expressed with matrices stored by row, and this version includes the new GxB_BY_ROW format. If you want the default format to be by row, just compile with -DBYROW, or add the following after calling GrB_init:

```
GxB_set (GxB_FORMAT, GxB_BY_ROW) ;
```

This BETA1 version has been fully tested and is ready to use, but the GxB_* extensions added to Version 2.1 may change as feedback from the GraphBLAS community is received. These include GxB_get, GxB_set, and GxB_AxB_METHOD, GxB_RANGE, GxB_STRIDE, and GxB_BACKWARDS, and their related definitions, described in Sections 4.9, 5, and 6.

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.

See [Dav18a] for a journal article on SuiteSparse:GraphBLAS, and [Dav18b] for a more recent conference paper. A full and precise definition of the GraphBLAS specification is provided in *The GraphBLAS C API Specification* by Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira, and Carl Yang [BMM+17a, BMM+17b], based on *GraphBLAS Mathematics* by Jeremy Kepner [Kep17]. The GraphBLAS C API Specification is available at `http://graphblas.org`. This version of SuiteSparse:GraphBLAS fully conforms to Version 1.2.0 (May 18, 2018) of that specification. In this User Guide, aspects of the GraphBLAS specification that would be true for any GraphBLAS implementation are simply called "GraphBLAS." Details unique to this particular implementation are referred to as SuiteSparse:GraphBLAS.

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

2 Basic Concepts

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

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

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

2.1 Graphs and sparse matrices

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

ports two kinds of sparse matrix formats: a regular sparse format, taking $O(n + |\mathbf{A}|)$ space, and a hypersparse format taking only $O(|\mathbf{A}|)$ space. As a result, creating a sparse matrix of size n-by-n where $n = 2^{60}$ (about 10^{18}) can be done on quite easily on a commodity laptop, limited only by $|\mathbf{A}|$.

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

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

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

2.2 Overview of GraphBLAS methods and operations

GraphBLAS provides a collection of *methods* to create, query, and free its of objects: sparse matrices, sparse vectors, types, operators, monoids, semirings, and a descriptor object used for parameter settings. Details are given in Section 4. 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 analog is given in the table below.

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

GraphBLAS can do far more than what MATLAB can do in these rough analogs, but the list provides a first step in describing what GraphBLAS can do. Details of each GraphBLAS operation are given in Section 7. With this brief overview, the full scope of GraphBLAS extensions of these operations can now be described.

GraphBLAS has 11 built-in scalar types: Boolean, single and double precision floating-point, 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 or vectors.

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 eight 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), and four logical operators: AND, OR, exclusive-OR, and Boolean equality. 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 960 unique semirings. With typecasting, any of these 960 semirings can be applied to matrices `C`, `A`, and `B` of any of the 11 types, in any combination. This gives $960 \times 11^3 = 1,277,760$ 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. All of the 1.3 million forms of matrix multiplication methods in SuiteSparse:GraphBLAS are typically just as fast as computing `C=A*B` in MATLAB using its own native sparse matrix multiplication methods, and sometimes faster.

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, 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 must 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 assign-

ment (`C(I,J)=A` where `I` and `J` are integer vectors), or 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 only $O(|\mathbf{A}|)$ 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 clear 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 4.9) 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_SCMP` 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_SCMP`)
            $\mathbf{C}$ is not modified
        else
            $\mathbf{C} = \mathbf{Z}$
    else
        if (`GrB_SCMP`)
            $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{Z}$
        else
            $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$

Both phases of the accum/mask process are illustrated in MATLAB notation in Figure 1. A GraphBLAS operation starts with its primary computation, producing a result `T`; for matrix multiply, `T=A*B`, or if `A` is transposed first, `T=A'*B`, for example. Applying the accumulator, mask (or its complement) to obtain the final result matrix `C` can be expressed in the MATLAB `accum_mask` function shown in the figure. This function is an exact, fully functional, and nearly-complete description of the GraphBLAS accumulator/mask operation. The only aspects it does not consider are typecasting (see Section 2.4), and the value of the implicit identity (for those, see another version in the `Test` folder).

One aspect of GraphBLAS cannot be as easily expressed in a MATLAB sparse matrix: namely, what is the implicit value of entries not in the pattern? To accommodate this difference in the `accum_mask` MATLAB function, each sparse matrix `A` is represented with its values `A.matrix` and its pattern, `A.pattern`. The latter could be expressed as the sparse matrix `A.pattern=spones(A)` or `A.pattern=(A~=0)` in MATLAB, if the implicit

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

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

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

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

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

value is zero. With different semirings, entries not in the pattern can be `1`, `+Inf`, `-Inf`, or whatever is the identity value of the monoid. As a result, Figure 1 performs its computations on two MATLAB matrices: the values in `A.matrix` and the pattern in the logical matrix `A.pattern`. Implicit values are untouched.

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

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

A set of MATLAB functions that precisely compute the $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$ operation according to the full GraphBLAS specification is provided in SuiteSparse:GraphBLAS as `GB_spec_accum.m`, which computes $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$, and `GB_spec_mask.m`, which computes $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$. SuiteSparse:GraphBLAS includes a complete list of `GB_spec_*` functions that illustrate every GraphBLAS operation; these are discussed in in Section 7.1.

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 very efficient since MATLAB supports sparse logical matrices. The Mask operator in GraphBLAS works identically as sparse logical indexing in MATLAB, and is equally as fast (or faster) in SuiteSparse:GraphBLAS.

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 3.4 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. In particular, a large integer outside the range of a smaller one is wrapped, modulo style. This differs from MATLAB.

However, unlike MATLAB, the C 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. 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`.

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

2.5 Notation and list of GraphBLAS operations

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

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

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

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

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

3 GraphBLAS Context and Sequence

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

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

The `GraphBLAS.h` file defines functions, types, and macros prefixed with `GrB_` and `GxB_` that may be used in user applications. The prefix `GrB_` denote items that appear in the official *GraphBLAS C API Specification*. The prefix `GxB_` refers to SuiteSparse-specific extensions to the GraphBLAS API. Both may be used in user applications but be aware that items with prefixes `GxB_` will not appear in other implementations of the GraphBLAS standard.

SPEC: The following macros are extensions to the spec.

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

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

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

#if GxB_IMPLEMENTATION > GxB_VERSION (1,4,0)
... use features from version 1.4.0 of a specific GraphBLAS implementation
#endif
```

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

Macro	purpose
`GxB_ABOUT`	this particular implementation, copyright, and URL
`GxB_DATE`	the date of this implementation
`GxB_SPEC`	the GraphBLAS specification for this implementation
`GxB_SPEC_DATE`	the date of the GraphBLAS specification
`GxB_LICENSE`	the license for this particular implementation

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

```
#define GxB_SUITESPARSE_GRAPHBLAS
```

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

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

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

The remainder of this section describes GraphBLAS functions that create, modify, and destroy the GraphBLAS context, or provide utility methods for dealing with errors:

GraphBLAS function	purpose	Section
`GrB_init`	start up GraphBLAS	3.1
`GrB_wait`	force completion of pending operations	3.2
`GrB_Info`	status code returned by GraphBLAS functions	3.3
`GrB_error`	get more details on the last error	3.4
`GrB_finalize`	finish GraphBLAS	3.5

3.1 GrB_init: initialize GraphBLAS

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

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

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

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

In addition, all methods and operations that extract values from a GraphBLAS object and return them into non-opaque user arrays always ensure that the computations for that object are completed when the method returns, namely: `GrB_*_nvals`, `GrB_*_extractElement`, `GrB_*_extractTuples`, and `GrB_*_reduce` (to scalar). These methods only ensure that the computations for a single object are completed. Use `GrB_wait` to ensure that all computations are completed (see Section 3.2).

SuiteSparse:GraphBLAS is not yet multithreaded, but it is safe to use in a multithreaded user application. This support is in Beta status, however, because of the behavior of `GrB_finalize` discussed below. User threads should not operate on the same matrices at the same time, with one exception. Multiple threads can use the same matrices as inputs to GraphBLAS operations, but only if they have no pending operations (use `GrB_Matrix_nvals` or `GrB_wait` first). User threads cannot simultaneously modify the output matrix of any GraphBLAS operation.

With multiple user threads, at least thread should call `GrB_init` before any thread calls any `GrB_*` or `GxB_*` function.

When the user application is finished, **all** user application threads that called any GraphBLAS operation must call `GrB_finalize`, to free up thread-local workspace allocated internally. This behavior is a deviation from the GraphBLAS API and may be modified in the future.

SPEC: Requiring all user threads in a multi-threaded user application to call `GrB_finalize` is specific to SuiteSparse:GraphBLAS and is a deviation from the *GraphBLAS C API Specification*.

3.2 GrB_wait: wait for pending operations to finish

```
GrB_Info GrB_wait ( ) ;     // finish all pending computations
```

`GrB_wait` forces all pending operations to complete. Blocking mode acts as if `GrB_wait` is called whenever a GraphBLAS method or operation returns to the user application.

Unless specific rules are followed, non-blocking mode can be unpredictable if user-defined functions have side effects or if they rely on global variables not under the control of GraphBLAS. Suppose the user application creates a user-defined operator that accesses a global variable. That operator is then used in a GraphBLAS operation, which is left pending. If the user application then changes the global variable before pending operations complete, the pending operations will be eventually computed with this different value.

Worse yet, a user-defined operator might be freed before it is needed to finish a pending operation. This causes undefined behavior.

For best results with GraphBLAS, user-defined functions should not have side effects, nor should they access global variables outside the control of GraphBLAS. This allows the non-blocking mode to be used at its fullest level of performance. However, both of these features can safely be used in user-defined functions if the following specific rules are followed.

- User-defined functions may be called in any order when used in a GraphBLAS operation. This order may change in non-obvious ways, even in the same GraphBLAS operation. For example, SuiteSparse:GraphBLAS relies on multiple algorithms for matrix multiplication, and selects between them automatically. The methods will call user-defined multiply and add operators in the semiring, in different order. The user application should not rely on any particular order used in a specific implementation of GraphBLAS.

- User-defined functions are permitted to access global variables. However, if they do so, the global variables they rely on should not be changed if any GraphBLAS methods or operations are still pending, assuming GraphBLAS is executing in non-blocking mode (see Section 3.1). To ensure this, the user application must call `GrB_wait` before changing any global variables relied upon by user-defined functions. Alternatively, computations can be forced to complete on selected matrices and vectors via `GrB_*_nvals`, `GrB_*_extractElement`,

GrB_*_extractTuples, and GrB_*_reduce (to scalar) applied to selected matrices and vectors. The GrB_*_nvals function is particularly well-suited for this purpose since it is otherwise an extremely lightweight computation in SuiteSparse:GraphBLAS.

- If any GraphBLAS methods or operations are still pending, freeing user-defined types, operators, monoids, semirings, vectors, matrices, or descriptors leads to undefined behavior. A user application must call GrB_wait before freeing any user-defined object, if a pending operation relies on it, or by selective completion via, say, GrB_*_nvals. Alternatively, if the user application is about to terminate GraphBLAS (see GrB_finalize below), then all GraphBLAS objects may be freed in any order, without calling GrB_wait. Pending computations will simply be abandoned.

GrB_wait ensures that all computations are completed for all objects. For specific objects, GrB_*_nvals, GrB_*_extractElement, GrB_*_extractTuples, and GrB_*_reduce (to scalar) ensure that the pending operations are completed just for the matrix or vector they operate on. No other GraphBLAS method or operation guarantees the completion of pending computations, even though they may happen to do so in any particular implementation. In the current version, SuiteSparse:GraphBLAS exploits the non-blocking mode in the GrB_*_setElement methods and the GrB_assign and GxB_subassign operations. Future versions of SuiteSparse:GraphBLAS may extend this to other methods and operations. Refer to the example at the end of Section 2.2.

If multiple user threads have created matrices or vectors, and those have pending operations, then a single call by one thread to GrB_wait causes all pending operations left by all threads to be completed. If other user threads are working on any of those matrices, this would result in a race condition. Therefore, GrB_wait should be called only when no other user thread is operating on any other matrix. Functions that cause a specific matrix to be finalized (GrB_*_nvals, GrB_*_extractElement, GrB_*_extractTuples, and GrB_*_reduce (to scalar)) can be safely called by multiple user threads on different matrices.

3.3 GrB_Info: status code returned by GraphBLAS

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

`GrB_SUCCESS`	the method or operation was successful
`GrB_NO_VALUE`	`A(i,j)` requested but not there. Its value is implicit.
`GrB_UNINITIALIZED_OBJECT`	object has not been initialized
`GrB_INVALID_OBJECT`	object is corrupted
`GrB_NULL_POINTER`	input pointer is `NULL`
`GrB_INVALID_VALUE`	generic error code; some value is bad
`GrB_INVALID_INDEX`	a row or column index is out of bounds; for indices passed as scalars, not in a list.
`GrB_DOMAIN_MISMATCH`	object domains are not compatible
`GrB_DIMENSION_MISMATCH`	matrix dimensions do not match
`GrB_OUTPUT_NOT_EMPTY`	output matrix already has values in it
`GrB_OUT_OF_MEMORY`	out of memory
`GrB_INSUFFICIENT_SPACE`	output array not large enough
`GrB_INDEX_OUT_OF_BOUNDS`	a row or column index is out of bounds; for indices in a list of indices.
`GrB_PANIC`	unrecoverable error. SuiteSparse:GraphBLAS never panics, however.

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

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

3.4 GrB_error: get more details on the last error

```
const char *GrB_error ( ) ;     // return a string describing the last error
```

Each GraphBLAS method and operation returns a `GrB_Info` error code. The `GrB_error` function returns additional information on the error in a thread-safe null-terminated string. The string returned by `GrB_error` is statically allocated in thread local storage and must not be freed or modified. Each user thread has its own error status. The simplest way to use it is just to print it out, such as:

```
info = GrB_some_method_here (...) ;
if (info != GrB_SUCCESS)
{
    printf ("%s\n", GrB_error ( )) ;
}
```

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

3.5 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`. There is no need to call `GrB_wait` in this case. When the user application is finished, **all** user threads must call `GrB_finalize`, to free up thread-local workspace allocated internally.

4 GraphBLAS Objects and their Methods

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

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

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

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

GraphBLAS provides 11 built-in types and 256 built-in operators. With these, 44 unique monoids and 960 unique semirings can be constructed.

SPEC: SuiteSparse:GraphBLAS predefines all unique monoids and semirings that can be constructed from built-in types and operators, as an extension to the spec. They appear in `GraphBLAS.h`. The `GxB_SelectOp` object is an extension to GraphBLAS.

4.1 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 eleven built-in types, and a user application can define any types of its own as well. The built-in types correspond to built-in types in C (`#include <stdbool.h>` and `#include <stdint.h>`), and the classes in MATLAB, as listed in the following table.

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

The user application can also define new types based on any `typedef` in the C language whose values are held in a contiguous region of memory. For example, a user-defined `GrB_Type` could be created to hold any C `struct` whose content is self-contained. A C `struct` containing pointers might be problematic because GraphBLAS would not know to dereference the pointers to traverse the entire "scalar" entry, but this can be done if the objects referenced by these pointers are not moved. A user-defined complex type with real and imaginary types can be defined, or even a "scalar" type containing a fixed-sized dense matrix (see Section 4.1.1). The possibilities are endless. GraphBLAS can create and operate on sparse matrices and vectors in any of these types, including any user-defined ones. For user-defined types, GraphBLAS simply moves the data around itself (via `memcpy`), and then passes the values back to user-defined functions when it needs to do any computations on the type. The next sections describe the methods for the `GrB_Type` object:

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

4.1.1 GrB_Type_new: **create a user-defined type**

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

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

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

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

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

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

Performing arithmetic on matrices and vectors with user-defined types requires operators to be defined. For example, the user application can define its own type for complex numbers, but then transposing the matrix with GraphBLAS will not compute the complex conjugate transpose. This corresponds to the array transpose in MATLAB (`C=A.'`) instead of the complex conjugate transpose (`C=A'`). To compute the complex conjugate transpose, the application would need to create a user-defined unary operator to conjugate a user-defined complex scalar, and then apply it to the matrix before or after the transpose, via `GrB_apply`. An extensive set of complex operators are provided in the `usercomplex.c` example in the `Demo` folder, along with an include file, `usercomplex.h`, that is suitable for inclusion in any user application. Thus, while GraphBLAS does not include any complex types or operators, SuiteSparse:GraphBLAS provides them in two simple "user" files in the `Demo` folder. Refer to Section 8.7 for more details on these two example user-defined types.

4.1.2 GxB_Type_size: return the size of a type

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

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

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

4.1.3 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.

Note the function signature of `GrB_Type_free`, above. It is illustrated with the generic name, `GrB_free`. Any GraphBLAS object can be freed with the single function, `GrB_free`. Refer to Section 4.10 for more 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`).

4.2 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.

There are six kinds of built-in unary operators: one, identity, additive inverse, absolute value, multiplicative inverse, and logical negation. In the notation in the table below, T is any of the 11 built-in types and is a place-holder for `BOOL`, `INT8`, `UINT8`, ... `FP32`, or `FP64`. For example, `GrB_AINV_INT32` is a unary operator that computes `z=-x` for two values `x` and `z` of type `GrB_INT32`.

The logical negation operator `GrB_LNOT` only works on Boolean types. The `GxB_LNOT_`T functions operate on inputs of type T, implicitly typecasting their input to Boolean and returning result of type T, with a value 1 for true and 0 for false. The operators `GxB_LNOT_BOOL` and `GrB_LNOT` are identical. Considering all combinations, there are thus 67 built-in unary operators ((6 kinds of operators) $\times$ (11 types), and `GrB_LNOT`).

GraphBLAS name	types (domains)	expression $z = f(x)$	description
`GxB_ONE_`T	$T \to T$	$z = 1$	one
`GrB_IDENTITY_`T	$T \to T$	$z = x$	identity
`GrB_AINV_`T	$T \to T$	$z = -x$	additive inverse
`GxB_ABS_`T	$T \to T$	$z = \|x\|$	absolute value
`GrB_MINV_`T	$T \to T$	$z = 1/x$	multiplicative inverse
`GxB_LNOT_`T	$T \to T$	$z = \neg(x \neq 0)$	logical negation
`GrB_LNOT`	`bool` $\to$ `bool`	$z = \neg x$	logical negation

SPEC: `GxB_ONE_`T, `GxB_ABS_`T and `GxB_LNOT_`T are extensions to the spec.

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

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

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

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

4.2.1 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
    const GrB_Type ztype,           // type of output z
    const 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. The pointers will be unique. That is, the user function is never called with pointers that point to the same space.

4.2.2 GxB_UnaryOp_ztype: return the type of z

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

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

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

4.2.3 GxB_UnaryOp_xtype: return the type of x

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

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

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

4.2.4 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.

4.3 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.

SuiteSparse:GraphBLAS has 17 kinds of built-in binary operators of the form $T \times T \rightarrow T$ that work on all 11 of the built-in types, T, for a total of 187 binary operators of this form. These are listed in the table below. For each of these operators, all domains (types) of the three operands are the same. The six comparison operators and three logical operators all return a result one for true and zero for false, in the same domain T as their inputs. These six comparison operators are useful as "multiply" operators for creating semirings with non-Boolean monoids.

GraphBLAS name	types (domains)	expression $z = f(x, y)$	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
`GrB_MIN_`T	$T \times T \rightarrow T$	$z = \min(x, y)$	minimum
`GrB_MAX_`T	$T \times T \rightarrow T$	$z = \max(x, y)$	maximum
`GrB_PLUS_`T	$T \times T \rightarrow T$	$z = x + y$	addition
`GrB_MINUS_`T	$T \times T \rightarrow T$	$z = x - y$	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_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
`GxB_ISGT_`T	$T \times T \rightarrow T$	$z = (x > y)$	greater than
`GxB_ISLT_`T	$T \times T \rightarrow T$	$z = (x < y)$	less than
`GxB_ISGE_`T	$T \times T \rightarrow T$	$z = (x \geq y)$	greater than or equal
`GxB_ISLE_`T	$T \times T \rightarrow T$	$z = (x \leq y)$	less than or equal
`GxB_LOR_`T	$T \times T \rightarrow T$	$z = (x \neq 0) \vee (y \neq 0)$	logical OR
`GxB_LAND_`T	$T \times T \rightarrow T$	$z = (x \neq 0) \wedge (y \neq 0)$	logical AND
`GxB_LXOR_`T	$T \times T \rightarrow T$	$z = (x \neq 0) \veebar (y \neq 0)$	logical XOR

SPEC: The `GxB_IS*_`T operators and the Boolean `GxB_L*_`T are extensions to the spec.

Another set of six kinds of built-in comparison operators have the form $T \times T \rightarrow$`bool`. They are defined for all eleven built-in types, for a total of 66 binary operators. Note that when T is `bool`, the six operators give the same

results as the six GxB_IS*_BOOL operators in the table above. These six comparison operators are useful as "multiply" operators for creating semirings with Boolean monoids.

GraphBLAS name	types (domains)	expression $z = f(x, y)$	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
GrB_GT_T	$T \times T \rightarrow$bool	$z = (x > y)$	greater than
GrB_LT_T	$T \times T \rightarrow$bool	$z = (x < y)$	less than
GrB_GE_T	$T \times T \rightarrow$bool	$z = (x \geq y)$	greater than or equal
GrB_LE_T	$T \times T \rightarrow$bool	$z = (x \leq y)$	less than or equal

Finally, GraphBLAS has three built-in binary operators that operate purely in the Boolean domain. These three are identical to the GxB_L*_BOOL operators described above, just with a shorter name.

GraphBLAS name	types (domains)	expression $z = f(x, y)$	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

This gives a total of 256 built-in binary operators listed in the tables above: 187 of the form $T \times T \rightarrow T$, 66 of the form $T \times T \rightarrow$ bool, and three purely Boolean. There are 240 unique operators since 16 of the 26 Boolean operators are redundant.

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

combined with a non-Boolean PLUS monoid to perform this operation.

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

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

Floating-point operations follow the IEEE 754 standard. Thus, computing $x/0$ for a floating-point x results in +Inf if x is positive, -Inf if x is negative, and NaN if x is zero. The application is not terminated. However, integer division by zero normally terminates an application. SuiteSparse:GraphBLAS avoids this by adopting the same rules as MATLAB, which are analogous to how the IEEE standard handles floating-point division by zero. For integers, when x is positive, $x/0$ is the largest positive integer, for negative x it is the minimum integer, and $0/0$ results in zero. For example, for an integer x of type GrB_INT32, $1/0$ is $2^{31} - 1$ and (-1)/0 is -2^{31}. Refer to Section 4.1 for a list of integer ranges.

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

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

4.3.1 GrB_BinaryOp_new: create a user-defined binary operator

```
GrB_Info GrB_BinaryOp_new
(
    GrB_BinaryOp *binaryop,         // handle for the new binary operator
    void *function,                 // pointer to the binary function
    const GrB_Type ztype,           // type of output z
    const GrB_Type xtype,           // type of input x
    const 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. The pointers will be unique. That is, the user function is never called with pointers that point to the same space.

4.3.2 GxB_BinaryOp_ztype: return the type of z

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

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

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

4.3.3 GxB_BinaryOp_xtype: return the type of x

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

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

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

4.3.4 GxB_BinaryOp_ytype: return the type of y

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

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

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

4.3.5 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.

4.4 SuiteSparse:GraphBLAS select operators: GxB_SelectOp

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

The `GxB_SelectOp` operator is used by `GxB_select` (see Section 7.13) to select entries from a matrix. Each entry `A(i,j)` is evaluated with the operator, which returns true if the entry is to be kept in the output, or false if it is not to appear in the output. The signature of the select function `f` is as follows:

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

There are five built-in select operators listed in the table below. For the first four operators, `k` is a pointer to a single scalar of type `int64_t`. Each operator can be used on any type, including user-defined types. User-defined select operators can also be created.

GraphBLAS name	MATLAB analog	description
`GxB_TRIL`	`C=tril(A,k)`	true for `A(i,j)` if `(j-i) <= k`
`GxB_TRIU`	`C=triu(A,k)`	true for `A(i,j)` if `(j-i) >= k`
`GxB_DIAG`	`C=diag(A,k)`	true for `A(i,j)` if `(j-i) == k`
`GxB_OFFDIAG`	`C=A-diag(A,k)`	true for `A(i,j)` if `(j-i) != k`
`GxB_NONZERO`	`C=A(A~=0)`	true if `A(i,j)` is nonzero

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

The built-in `GxB_NONZERO` select operator is unique in that it is a function of the value of the entry a_{ij}, but it is still type-generic. It does this by simply returning false if all bits in the value are zero, or true otherwise. This gives the proper result for any built-in type, since integer and floating-point zeros

are represented this way. For user-defined types, the function returns the same thing. This action is well-defined but its suitability for any particular user-defined type must be determined according to how the user application defines the type, and what a value with all bits zero means for this type. Whatever it means, if the bits of a value with a user-defined type are all zero, the function returns false, and if any bit is one, the `GxB_NONZERO` function returns true.

The following methods operate on the `GxB_SelectOp` object:

`GxB_SelectOp_new`	create a user-defined select operator
`GxB_SelectOp_xtype`	return the type of the input x
`GxB_SelectOp_free`	free a user-defined select operator

4.4.1 GxB_SelectOp_new: create a user-defined select operator

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

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

The `function` argument to `GxB_SelectOp_new` is a pointer to a user-defined function whose signature is given at the beginning of Section 4.4. Given the properties of an entry a_{ij} in an m-by-n matrix, the `function` should return `true` if the entry should be kept in the output of `GxB_select`, or `false` if it should not appear in the output.

The type `xtype` is the GraphBLAS type of the input x of the user-defined function $z = f(i, j, m, n, x, k)$. The type may be built-in or user-defined, or it may even be `GrB_NULL`. If the `xtype` is `GrB_NULL`, then `GxB_select` does not pass the value of $x = a_{ij}$ to the select function, but passes `GrB_NULL` for the input `x` to the user-defined select `function`.

For example, to delete all entries in a matrix equal to the additive identity of a monoid, create a user-defined `GxB_SelectOp` with a function that compares its input value `x` with the value of the identity, passed in to it via the `k` parameter of type `void *`. The user-defined function should return `false` if `x` is equal the value `(*k)`, and false otherwise.

4.4.2 GxB_SelectOp_xtype: return the type of x

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

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

4.4.3 GxB_SelectOp_free: free a user-defined select operator

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

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

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

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

4.5 GraphBLAS monoids: GrB_Monoid

A *monoid* is defined on a single domain (that is, a single type), T. It consists of an associative binary operator $z = f(x, y)$ whose three operands x, y, and z are all in this same domain T (that is $T \times T \to T$). The associative operator must also have an identity element, or "zero" in this domain, such that $f(x, 0) = f(0, x) = 0$. 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.

Four kinds of built-in operators (`MIN`, `MAX`, `PLUS`, `TIMES`) can be used to form monoids for each of the ten non-Boolean built-in types, and 12 can be used for Boolean monoids, all of which are listed in the table below. This is a total of 52 valid monoids that can be constructed from built-in types and operators, although 8 of the 12 Boolean monoids are redundant (the four remaining being `OR`, `AND`, `XOR`, and `EQ`). There are thus a total of 44 unique monoids that can be constructed using built-in binary operators. Since the built-in monoids are also commutative, all of them can be used to create a semiring. Recall that a commutative operator $f(x, y)$ is one for which the condition $f(a, b) = f(b, a)$ always holds. That is, the two operands can be swapped and the results remain the same. One of the components of a semiring is a commutative monoid.

GraphBLAS name	types (domains)	expression $z = f(x, y)$	identity
`GrB_MIN_`T	$T \times T \to T$	$z = \min(x, y)$	$+\infty$
`GrB_MAX_`T	$T \times T \to T$	$z = \max(x, y)$	$-\infty$
`GrB_PLUS_`T	$T \times T \to T$	$z = x + y$	0
`GrB_TIMES_`T	$T \times T \to T$	$z = xy$	1
`GrB_LOR`, `GxB_LOR_BOOL`	`bool` $\times$ `bool` $\to$ `bool`	$z = x \vee y$	false
`GrB_LAND`, `GxB_LAND_BOOL`	`bool` $\times$ `bool` $\to$ `bool`	$z = x \wedge y$	true
`GrB_LXOR`, `GxB_LXOR_BOOL`	`bool` $\times$ `bool` $\to$ `bool`	$z = x \veebar y$	false
`GrB_EQ_BOOL`, `GxB_ISEQ_BOOL`	`bool` $\times$ `bool` $\to$ `bool`	$z = (x == y)$	true

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

`GrB_Monoid_new`	create a monoid
`GxB_Monoid_operator`	return the monoid operator
`GxB_Monoid_identity`	return the monoid identity value
`GrB_Monoid_free`	free a monoid

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

4.5.1 GrB_Monoid_new: create a monoid

```
GrB_Info GrB_Monoid_new             // create a monoid
(
    GrB_Monoid *monoid,             // handle of monoid to create
    const GrB_BinaryOp op,          // binary operator of the monoid
    const <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 type-generic place-holder. For built-in types, it is the C type corresponding to the built-in type (see Section 4.1), such as `bool`, `int32_t`, `float`, or `double`. In this case, `identity` is a `const` 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`.

4.5.2 GxB_Monoid_operator: return the monoid operator

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

`GxB_Monoid_operator` returns the binary operator of the monoid.

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

4.5.3 GxB_Monoid_identity: return the monoid identity

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

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

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

4.5.4 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.

4.6 GraphBLAS semirings: GrB_Semiring

A *semiring* defines all the operators required to define the multiplication of two sparse matrices in GraphBLAS, $\mathbf{C} = \mathbf{AB}$. The "add" operator is a commutative and associative monoid, and the binary "multiply" operator defines a function $z = fmult(x, y)$ where the type of z matches the exactly with the monoid type. SuiteSparse:GraphBLAS includes 960 predefined built-in semirings, which are all those that can be constructed from built-in types and operators. The next sections define the following methods for the `GrB_Semiring` object:

`GrB_Semiring_new`	create a semiring
`GxB_Semiring_add`	return the additive monoid of a semiring
`GxB_Semiring_multipliy`	return the binary operator of a semiring
`GrB_Semiring_free`	free a semiring

4.6.1 GrB_Semiring_new: create a semiring

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

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

Using built-in types and operators, 960 unique semirings can be built. This count excludes redundant Boolean operators (for example `GrB_TIMES_BOOL` and `GxB_LAND_BOOL` are different operators but they are redundant since they always return the same result).

- 680 semirings with a multiplier $T \times T \to T$ where T is non-Boolean, from the complete cross product of:
  - 4 add monoids (`MIN`, `MAX`, `PLUS`, `TIMES`)

  - 17 multiply operators (`FIRST`, `SECOND`, `MIN`, `MAX`, `PLUS`, `MINUS`, `TIMES`, `DIV`, `ISEQ`, `ISNE`, `ISGT`, `ISLT`, `ISGE`, `ISLE`, `LOR`, `LAND`, `LXOR`)
  - 10 non-Boolean types, T

- 240 semirings with a comparison operator $T \times T \rightarrow$ `bool`, where T is non-Boolean, from the complete cross product of:
  - 4 Boolean add monoids (`LAND`, `LOR`, `LXOR`, `EQ`)
  - 6 multiply operators (`EQ`, `NE`, `GT`, `LT`, `GE`, `LE`)
  - 10 non-Boolean types, T

- 40 semirings with purely Boolean types, `bool` $\times$ `bool` $\rightarrow$ `bool`, from the complete cross product of:
  - 4 Boolean add monoids (`LAND`, `LOR`, `LXOR`, `EQ`)
  - 10 multiply operators (`FIRST`, `SECOND`, `LOR`, `LAND`, `LXOR`, `EQ`, `GT`, `LT`, `GE`, `LE`)

SPEC: SuiteSparse:GraphBLAS pre-defines all 960 semirings that can be constructed from built-in types and operators, as an extension to the spec.

4.6.2 GxB_Semiring_add: return the additive monoid of a semiring

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

`GxB_Semiring_add` returns the additive monoid of a semiring.

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

4.6.3 GxB_Semiring_multiply: return multiply operator of a semiring

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

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

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

4.6.4 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.

4.7 GraphBLAS vectors: GrB_Vector

Many of the methods for GraphBLAS vectors require a row index or a size. Many methods for matrices require both a row and column index, or a row and column dimension. These are all integers of a specific type, `GrB_Index`, which is defined in `GraphBLAS.h` as

```
typedef uint64_t GrB_Index ;
```

Row and column indices of an `nrows`-by-`ncols` matrix range from zero to the `nrows-1` for the rows, and zero to `ncols-1` for the columns. Indices are zero-based, like C, and not one-based, like MATLAB. In SuiteSparse:GraphBLAS, the largest size permitted for any integer of `GrB_Index` is 2^{60}. The largest `GrB_Matrix` that SuiteSparse:GraphBLAS can construct is thus 2^{60}-by-2^{60}. An n-by-n matrix A that size can easily be constructed in practice with $O(|\mathbf{A}|)$ memory requirements, where $|\mathbf{A}|$ denotes the number of entries that explicitly appear in the pattern of $\mathbf{A}$. The time and memory required to construct a matrix that large does not depend on n, since SuiteSparse:GraphBLAS can represent $\mathbf{A}$ in hypersparse form (see Section 5.2). The largest `GrB_Vector` that can be constructed is 2^{60}-by-1.

If compiled for use in MATLAB, this maximum size is reduced to match the MATLAB maximum index size, which is $2^{48} - 1$.

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

`GrB_Vector_new`	create a vector
`GrB_Vector_dup`	copy a vector
`GrB_Vector_clear`	clear a vector of all entries
`GrB_Vector_size`	return the size of a vector
`GrB_Vector_nvals`	return the number of entries in a vector
`GxB_Vector_type`	return the type of a vector
`GrB_Vector_build`	build a vector from a set of tuples
`GrB_Vector_setElement`	add a single entry to a vector
`GrB_Vector_extractElement`	get a single entry from a vector
`GrB_Vector_extractTuples`	get all entries from a vector
`GxB_Vector_resize`	resize a vector
`GrB_Vector_free`	free a vector

4.7.1 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
    const GrB_Type type,    // type of vector to create
    const GrB_Index n       // vector dimension is n-by-1
) ;
```

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

4.7.2 GrB_Vector_dup: copy a vector

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

`GrB_Vector_dup` makes a deep copy of a sparse vector, like `w=u` in MATLAB. In GraphBLAS, it is possible, and valid, to write the following:

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

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

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

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

4.7.3 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 (:) = 0` in MATLAB. The type and dimension of `v` do not change. In SuiteSparse:GraphBLAS, any pending updates to the vector are discarded.

4.7.4 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.

4.7.5 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 explicit entries" in GraphBLAS.

Forced completion: All computations for the vector `v` are guaranteed to be finished when `GrB_Vector_nvals` method returns. That is, it acts like an object-specific `GrB_wait` for just this particular vector `v`, which is a side-effect useful in its own right. For example, suppose the computations required for `v` rely upon a user-defined operator that accesses a user-controlled global variable outside the scope or control of GraphBLAS. If the user-application needs to modify or free the variable, `GrB_Vector_nvals` can be used to force all pending operations for this vector `v` to complete. The user application can then safely modify the global variable. A call to `GrB_Vector_nvals(&nvals,v)` only ensures that the computations require to compute `v` are finished; other pending computations for other objects may remain. To ensure that all pending computations are complete for all GraphBLAS objects, use `GrB_wait` instead.

4.7.6 GxB_Vector_type: return the type of a vector

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

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

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

4.7.7 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
    const 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 4.8.8), 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.

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

4.7.8 GrB_Vector_setElement: **add a single entry to a vector**

```
GrB_Info GrB_Vector_setElement          // w(i) = x
(
    GrB_Vector w,                       // vector to modify
    const <type> x,                     // scalar to assign to w(i)
    const GrB_Index i                   // row 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 4.8.9.

4.7.9 GrB_Vector_extractElement: get a single entry from a vector

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

`GrB_Vector_extractElement` extracts a single entry from a vector, `x = v(i)`. The method is identical to extracting a single entry `x = A(i,0)` from an `n`-by-1 matrix, so further details of this method are discussed in Section 4.8.10, which discusses `GrB_Matrix_extractElement`. In this case, the column index is implicitly `j=0`.

Forced completion: All computations for the vector `v` are guaranteed to be finished when the method returns.

4.7.10 GrB_Vector_extractTuples: get all entries from a vector

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

`GrB_Vector_extractTuples` extracts all tuples from a sparse vector, analogous to `[I,~,X] = find(v)` in MATLAB. This function is identical to its `GrB_Matrix_extractTuples` counterpart, except that the array of column indices `J` does not appear in this function. Refer to Section 4.8.11 where further details of this function are described.

Forced completion: All computations for the vector `v` are guaranteed to be finished when the method returns.

4.7.11 GxB_Vector_resize: resize a vector

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

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

4.7.12 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. In SuiteSparse:GraphBLAS, any pending updates to the vector are abandoned.

4.8 GraphBLAS matrices: GrB_Matrix

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

`GrB_Matrix_new`	create a matrix
`GrB_Matrix_dup`	copy a matrix
`GrB_Matrix_clear`	clear a matrix of all entries
`GrB_Matrix_nrows`	return the number of rows of a matrix
`GrB_Matrix_ncols`	return the number of columns of a matrix
`GrB_Matrix_nvals`	return the number of entries in a matrix
`GxB_Matrix_type`	return the type of a matrix
`GrB_Matrix_build`	build a matrix from a set of tuples
`GrB_Matrix_setElement`	add a single entry to a matrix
`GrB_Matrix_extractElement`	get a single entry from a matrix
`GrB_Matrix_extractTuples`	get all entries from a matrix
`GxB_Matrix_resize`	resize a matrix
`GrB_Matrix_free`	free a matrix

4.8.1 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
    const GrB_Type type,    // type of matrix to create
    const GrB_Index nrows,  // matrix dimension is nrows-by-ncols
    const 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.

4.8.2 GrB_Matrix_dup: copy a matrix

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

`GrB_Matrix_dup` makes a deep copy of a sparse matrix, like `C=A` in MATLAB. In GraphBLAS, it is possible, and valid, to write the following:

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

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

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

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

4.8.3 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. In SuiteSparse:GraphBLAS, any pending updates to the matrix are discarded.

4.8.4 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 (`nrow=size(A,1)` in MATLAB).

4.8.5 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).

4.8.6 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, like `nnz(A)` in MATLAB.

Forced completion: All computations for the matrix `A` are guaranteed to be finished when the method returns. That is, it acts like an object-specific `GrB_wait` for just this particular matrix `A`. Other pending computations for other objects may remain. To ensure that all pending computations are complete for all GraphBLAS objects, used `GrB_wait` instead.

4.8.7 GxB_Matrix_type: return the 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
) ;
```

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

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

4.8.8 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
    const GrB_Index nvals,          // number of tuples
    const GrB_BinaryOp dup          // binary function to assemble duplicates
) ;
```

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

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

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

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

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

in positions k1, k2, k3, and k4 in [I,J,X], where k1 < k2 < k3 < k4, then the following operations will occur in order:

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

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

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

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

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

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

4.8.9 GrB_Matrix_setElement: add a single entry to a matrix

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

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

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

The scalar `x` is typecasted into the type of `C`. Any value can be passed to this function and its type will be detected, via the `_Generic` feature of ANSI C11. For a user-defined type, `x` is a `void *` pointer that points to a memory space holding a single entry of this user-defined type. This user-defined type must exactly match the user-defined type of `C` since no typecasting is done between user-defined types.

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

needed in another computation, or whenever GrB_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. Sections 7.11 gives more details on which GxB_subassign and GrB_assign operations can be interleaved with calls to setElement without forcing updates to be assembled. Other methods that do not access the existing entries may also be done without forcing the updates to be assembled, namely GrB_Matrix_clear (which erases all pending updates), GrB_Matrix_free, GrB_Matrix_ncols, GrB_Matrix_nrows, GxB_Matrix_type, and of course GrB_Matrix_setElement itself. All other methods and operations cause the updates to be assembled. Future versions of SuiteSparse:GraphBLAS may extend this list.

See Section 8.3 for an example of how to use GrB_Matrix_setElement.

4.8.10 GrB_Matrix_extractElement: get a single entry from a matrix

```
GrB_Info GrB_Matrix_extractElement      // x = A(i,j)
(
    <type> *x,                          // extracted scalar
    const GrB_Matrix A,                 // matrix to extract a scalar from
    const GrB_Index i,                  // row index
    const 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 not present then GraphBLAS does not know its value, since its value depends on the implicit value, which is the identity value of the additive monoid of the semiring. It is not a characteristic of the matrix itself, but of the semiring it is used in. A matrix can be used in any compatible semiring, and even a mixture of semirings, so the implicit value can change as the semiring changes.

As a result, if the entry is present, `x=A(i,j)` is performed and the scalar `x` is returned with this value. The method returns `GrB_SUCCESS`. If the entry is not present, `x` is not modified, and `GrB_NO_VALUE` is returned to the caller. What this means is up to the caller.

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

Forced completion: All computations for the matrix `A` are guaranteed to be finished when the method returns. In particular, 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.

4.8.11 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. SuiteSparse:GraphBLAS chooses to always return them in sorted order, depending on whether the matrix is stored by row or by column.

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.

Forced completion: All computations for the matrix A are guaranteed to be finished when the method returns.

4.8.12 GxB_Matrix_resize: resize a matrix

```
GrB_Info GxB_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
) ;
```

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

4.8.13 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. In SuiteSparse:GraphBLAS, any pending updates to the matrix are abandoned.

4.9 GraphBLAS descriptors: GrB_Descriptor

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

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

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,       // descriptor for output of a method
    GrB_MASK,       // descriptor for the mask input of a method
    GrB_INP0,       // descriptor for the first input of a method
    GrB_INP1,       // descriptor for the second input of a method
    GxB_AxB_METHOD  // descriptor for selecting C=A*B algorithm
}
GrB_Desc_Field ;
```

```
typedef enum
{
    // for all GrB_Descriptor fields:
    GxB_DEFAULT,        // default behavior of the method

    // for GrB_OUTP only:
    GrB_REPLACE,        // clear the output before assigning new values to it

    // for GrB_MASK only:
    GrB_SCMP,           // use the structural complement of the input

    // for GrB_INP0 and GrB_INP1 only:
    GrB_TRAN,           // use the transpose of the input

    // for GxB_AxB_METHOD only:
    GxB_AxB_GUSTAVSON,  // gather-scatter saxpy method
    GxB_AxB_HEAP,       // heap-based saxpy method
    GxB_AxB_DOT         // dot product
}
GrB_Desc_Value ;
```

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

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

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

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

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

  If the `desc [GrB_MASK]` parameter is set to `GrB_SCMP`, 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.

  Using a parameter to complement the `Mask` is very useful because constructing the actual complement of a very sparse mask is impossible since it has too many entries. If the number of places in `C` that should

be modified is very small, then use a sparse mask without complementing it. If the number of places in `C` that should be protected from modification is very small, then use a sparse mask to indicate those places, and use a descriptor `GrB_MASK` that complements the use of the mask.

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

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

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

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

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

  - `GxB_DEFAULT` means that a method is selected automatically.
  - `GxB_AxB_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)`). It requires workspace of size m, to the number of rows of `C`, which is not suitable if the matrices are extremely sparse.
  - `GxB_AxB_HEAP`: a heap-based method, computing `C(:,j)=A*B(:,j)` via a heap of size equal to the maximum number of entries in any column of `B`. The method is very good for hypersparse matrices,

particularly when $|\mathbf{B}|$ is less than the number of rows of `C`. The method used is similar to Algorithm II in [BG08] (see also [BG12]). It computes `C` in the same order as Gustavson's method, using a heap instead of a large gather/scatter workspace. The heap has size b, equal to the maximum number of entries in any one vector of `B`.

– `GxB_AxB_DOT`: computes `C(i,j)=A(:,i)'*B(:,j)`, for each entry `C(i,j)`. If the mask is present and not complemented, only entries for which `M(i,j)=1` are computed. This is a very specialized method that works well only if the mask is present, very sparse, and not complemented, or when `C` is tiny. For example, it works very well when `A` and `B` are tall and thin, and `C<M>=A'*B` or `C=A'*B` are computed. It 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 can work either without any workspace at all, or with workspace of size $O(k)$, where k is the inner dimension (the number of rows of `B` if the matrices are stored by column). It does not need any workspace at all if `A` or `B` are completely dense, and is the fastest method in this case. Otherwise, the method is faster if it has access to workspace, and thus it uses $O(k)$ workspace if $k < (|\mathbf{A}| + |\mathbf{B}|)$. If this condition does not hold, then the workspace could dominate the memory usage, so it foregoes the use of any workspace. Since it uses no workspace if $k \geq (|\mathbf{A}| + |\mathbf{B}|)$, it can work very well for extremely sparse or hypersparse matrices, when the mask is present and not complemented.

4.9.1 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, mask is not complemented, neither input matrix is transposed, and the method used in `C=A*B` is selected automatically).

4.9.2 GrB_Descriptor_set: set a parameter in a descriptor

```
GrB_Info GrB_Descriptor_set     // set a parameter in a descriptor
(
    GrB_Descriptor desc,        // descriptor to modify
    const GrB_Desc_Field field, // parameter to change
    const GrB_Desc_Value val    // value to change it to
) ;
```

`GrB_Descriptor_set` sets a descriptor field (`GrB_OUTP`, `GrB_MASK`, `GrB_INP0`, `GrB_INP1`, or `GxB_AxB_METHOD`) to a particular value (`GxB_DEFAULT`, `GrB_SCMP`, `GrB_TRAN`, `GrB_REPLACE`, `GxB_AxB_GUSTAVSON`, `GxB_AxB_HEAP`, or `GxB_AxB_DOT`).

Descriptor field	Default	Non-default
`GrB_OUTP`	`GxB_DEFAULT`: The output matrix is not cleared. The operation computes $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$.	`GrB_REPLACE`: After computing $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$, the output $\mathbf{C}$ is cleared of all entries. Then $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ is performed.
`GrB_MASK`	`GxB_DEFAULT`: The Mask is not complemented. `Mask(i,j)=1` means the value C_{ij} can be modified by the operation, while `Mask(i,j)=0` means the value C_{ij} shall not be modified by the operation.	`GrB_SCMP`: 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_INP0`	`GxB_DEFAULT`: The first input is not transposed prior to using it in the operation.	`GrB_TRAN`: The first input is transposed prior to using it in the operation. Only matrices are transposed, never vectors.
`GrB_INP1`	`GxB_DEFAULT`: The second input is not transposed prior to using it in the operation.	`GrB_TRAN`: The second input is transposed prior to using it in the operation. Only matrices are transposed, never vectors.
`GrB_AxB_METHOD`	`GxB_DEFAULT`: The method used for computing `C=A*B` is selected automatically.	`GxB_AxB_`*method*: The selected method is used to compute `C=A*B`.

4.9.3 GxB_Desc_set: set a parameter in a descriptor

```
GrB_Info GxB_Desc_set           // set a parameter in a descriptor
(
    GrB_Descriptor desc,        // descriptor to modify
    const GrB_Desc_Field field, // parameter to change
    ...                         // value to change it to
) ;
```

`GxB_Desc_set` is identical to `GrB_Descriptor_set`, except that the type of the third parameter can vary with the field. All descriptor fields are currently of type `GrB_Desc_Value`, so currently this function is identical in all ways to `GrB_Descriptor_set`, except for the name of the function. Future versions of this function will allow for arbitrary types of the third parameter, depending on the field. For a simpler-to-use alternative, see `GxB_set` described in Section 5.

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

4.9.4 GxB_Desc_get: get a parameter from a descriptor

```
GrB_Info GxB_Desc_get           // get a parameter from a descriptor
(
    const GrB_Descriptor desc,  // descriptor to query; NULL means defaults
    const GrB_Desc_Field field, // parameter to query
    ...                         // value of the parameter
) ;
```

`GxB_Desc_get` returns the value of a single field in a descriptor. The type of the third parameter is a pointer to a variable type, whose type depends on the field. Currently, all descriptor values are of type `GrB_Desc_Value`, so this third parameter is a pointer to a scalar value of type `GrB_Desc_Value`. For a simpler-to-use alternative, see `GxB_get` described in Section 5.

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

4.9.5 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.

There are currently no predefined descriptors, but if these are added in the future, this function will do nothing if passed a built-in descriptor.

4.10 GrB_free: free any GraphBLAS object

Each of the nine 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 nine objects. If `G` is any of the nine 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. In all cases, the `GrB_free` function in SuiteSparse:GraphBLAS always returns `GrB_SUCCESS`.

5 SuiteSparse:GraphBLAS Options

SPEC: GxB_set and GxB_get are extensions to the specification.

SuiteSparse:GraphBLAS includes two type-generic methods, `GxB_set` and `GxB_get`, that set and query various options and parameters settings, including a generic way to set values in the `GrB_Descriptor` object. Using these methods, the user application can provide hints to SuiteSparse:GraphBLAS on how it should store and operate on its matrices. These hints have no effect on the results of any GraphBLAS operation (except perhaps floating-point roundoff differences), but they can have a great impact on the amount of time or memory taken.

- `GxB_set (field, value)` provides hints to SuiteSparse:GraphBLAS on how it should store all matrices created after calling this function: by row, by column, and whether or not to use a *hypersparse* format [BG08, BG12]. These are global options that modify all matrices created after calling this method.

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

- `GxB_set (GrB_Descriptor desc, field, value)` is another way to set the value of a field in a `GrB_Descriptor`. It is identical to `GrB_Descriptor_set`, just with a generic name. Except for `GxB_AxB_METHOD`, the descriptor settings are not hints but are always implemented by GraphBLAS, since they affect the results of any GraphBLAS operation.

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

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

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

5.1 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.

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, since the result is already held in the data structure a single list, the jth column vector. However, `w=A(i,:)` is very slow in MATLAB, since every column in the matrix has to be searched to see if it contains row `i`. In MATLAB, if many such accesses are made, it is much better to transpose the matrix (say `AT=A'`) and then use `w=AT(:,i)` instead. This can have a dramatic impact on the performance of MATLAB.

Likewise, if `u` is a very sparse column vector and `A` is stored by column, then `w=u'*A` (via `GrB_vxm`) is slower than `w=A*u` (via `GrB_mxv`). The opposite is true if the matrix is stored by row.

An example of this can be found in Section B.1 of Version 1.2 of the GraphBLAS API Specification, where the breadth-first search `BFS` uses `GrB_vxm` to compute `q'=q'*A`. This method is not fast if the matrix `A` is stored

by column. The `bfs5` and `bfs6` examples in the `Demo/` folder of SuiteSparse:GraphBLAS use `GrB_mxv` instead, because in Version 2.0.3 and earlier, SuiteSparse:GraphBLAS only supported a data structure that stored its matrices by column. The methods are equivalent; they just use a matrix `A` that is the transpose of the other.

SuiteSparse:GraphBLAS still stores its sparse matrices by column, by default, just like MATLAB. However, it can also be instructed to store any selected matrices, or all matrices, by row instead, so that `w=A(i,:)` (via `GrB_Col_extract` with a transposed descriptor) is very fast. This allows the BFS example in GraphBLAS API to be done efficiently. The change in data format has no effect on the result, just the time and memory usage. To use a row-oriented format by default, the following can be done in a user application that tends to access its matrices by row.

```
GrB_init (...) ;
// just after GrB_init: do the following:
#ifdef GxB_SUITESPARSE_GRAPHBLAS
GxB_set (GxB_FORMAT, GxB_BY_ROW) ;
#endif
```

If this is done, and no other `GxB_set` calls are made with `GxB_FORMAT`, all matrices will be stored by row. Alternatively, SuiteSparse:GraphBLAS can be compiled with `-DBYROW`, which changes the default format to `GxB_BY_ROW`, with no calls to any `GxB_*` function.

5.2 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 $k \leq n$ vectors in the data structure. One advantage of the hypersparse structure is the memory required for an `m`-by-`n` hypersparse CSC matrix is only $O(|\mathbf{A}|)$, independent of `m` and `n`. Algorithms that must visit all non-empty columns of a matrix are much faster when working with hypersparse matrices, since empty columns can be skipped.

The `hyper_ratio` 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_ratio` = `GxB_HYPER_DEFAULT`, which is an `extern const double` value, currently set to 0.0625. This default ratio may change in the future.

The `hyper_ratio` 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_ratio`.

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_ratio` is assigned at startup by `GrB_init`, and can then be modified globally with `GxB_set`. All new matrices are created with the same `hyper_ratio`, 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_ratio = 0.2 ;
GxB_set (A, GxB_HYPER, hyper_ratio) ;
```

To force a matrix to always be non-hypersparse, use `hyper_ratio` equal

to GxB_NEVER_HYPER. To force a matrix to always stay hypersparse, set hyper_ratio 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_ratio value. The matrix is created in in either GxB_BY_ROW or GxB_BY_COL format, as determined by the last call to GxB_set(GxB_FORMAT,...) or GrB_init.

A new matrix C created via GrB_dup (&C,A) inherits the CSR/CSC format, hypersparsity format, and hyper_ratio from A.

Parameter types: The GxB_Option_Field enumerated type gives the type of the field parameter for the second argument of GxB_set and GxB_get, for setting global options or matrix options.

```
typedef enum
{
    GxB_HYPER,      // defines switch to hypersparse format (a double value)
    GxB_FORMAT      // defines CSR/CSC format: GxB_BY_ROW or GxB_BY_COL
}
GxB_Option_Field ;
```

The GxB_FORMAT field can be by row or by column, set to a value with the type GxB_Format_Value:

```
typedef enum
{
    GxB_BY_ROW,         // CSR: compressed sparse row format
    GxB_BY_COL          // CSC: compressed sparse column format
}
GxB_Format_Value ;
```

The default format is by column, just like MATLAB. This is given by the predefined value GxB_FORMAT_DEFAULT, which is equal to GxB_BY_COL if default compile-time options are used. To change the default at compile time to GxB_BY_ROW, compile the SuiteSparse:GraphBLAS library with -DBYROW. This changes GxB_FORMAT_DEFAULT to GxB_BY_ROW. The default hypersparsity ratio is 0.0625 (1/16), but this value may change in the future.

Setting the GxB_HYPER field to GxB_ALWAYS_HYPER ensures a matrix always stays hypersparse. If set to GxB_NEVER_HYPER, it always stays non-hypersparse. At startup, GrB_init defines the following initial settings:

```
GxB_set (GxB_HYPER, GxB_HYPER_DEFAULT) ;
GxB_set (GxB_FORMAT, GxB_FORMAT_DEFAULT) ;
```

That is, by default, all new matrices are held by column in CSC format, unless `-DBYROW` is used at compile time, in which case the default is to store all new matrices by row in CSR format. If a matrix has fewer than $n/16$ columns, it can be converted to hypersparse format. If it has more than $n/8$ columns, it can be converted to non-hypersparse format. These options can be changed for all future matrices with `GxB_set`. For example, to change all future matrices to be in non-hypersparse CSR when created, use:

```
GxB_set (GxB_HYPER, GxB_NEVER_HYPER) ;
GxB_set (GxB_FORMAT, GxB_BY_ROW) ;
```

Then if a particular matrix needs a different format, then (as an example):

```
GxB_set (A, GxB_HYPER, 0.1) ;
GxB_set (A, GxB_FORMAT, GxB_BY_COL) ;
```

This changes the matrix `A` so that it is stored by column, and it is converted from non-hypersparse to hypersparse format if it has fewer than 10% non-empty columns. If it is hypersparse, it is a candidate for conversion to non-hypersparse if has 20% or more non-empty columns. If it has between 10% and 20% non-empty columns, it remains in whatever format it is currently in.

MATLAB only supports a non-hypersparse CSC format. The format in SuiteSparse:GraphBLAS that is equivalent to the MATLAB format is given below:

```
GrB_init ( ) ;
GxB_set (GxB_HYPER, GxB_NEVER_HYPER) ;
GxB_set (GxB_FORMAT, GxB_BY_COL) ;
// no subsequent use of GxB_HYPER or GxB_FORMAT
```

The `GxB_HYPER` and `GxB_FORMAT` options should be considered as suggestions from the user application as to how SuiteSparse:GraphBLAS can obtain the best performance for a particular application. SuiteSparse:GraphBLAS is free to ignore any of these suggestions, both now and in the future, and the available options and formats may be augmented in the future. Any prior options no longer needed in future versions of SuiteSparse:GraphBLAS will be silently ignored, so the use these options is safe for future updates.

SuiteSparse:GraphBLAS is not yet multi-threaded, but it is thread-safe if the user application threads do not operate on the same matrices at the same time. Output matrices and vectors used by different threads must be different, and input matrices and vectors can be safely used only if any pending computations on them have finished, via `GrB_wait` or the per-matrix methods, `GrB_*_nvals`, `GrB_*_extractElement`, `GrB_*_extractTuples`, and reduction to a scalar via `GrB_*_reduce`.

All threads in the same user application share the same global options, including hypersparsity and CSR/CSC format determined by `GxB_set`, and blocking mode determined by `GrB_init`. Specific format and hypersparsity parameters of each matrix are specific to that matrix and can be independently changed.

5.3 GxB_set: set a global option

```
GrB_Info GxB_set                    // set a global default option
(
    const GxB_Option_Field field,   // option to change
    ...                             // value to change it to
) ;
```

This usage of `GxB_set` sets the value of a global option. The `field` parameter can be `GxB_HYPER` or `GxB_FORMAT`.

For example, the following usage sets the global hypersparsity ratio to 0.2, and the format of future matrices to `GxB_BY_ROW`. No existing matrices are changed.

```
GxB_set (GxB_HYPER, 0.2) ;
GxB_set (GxB_FORMAT, GxB_BY_ROW) ;
```

5.4 GxB_set: set a matrix option

```
GrB_Info GxB_set                    // set an option in a matrix
(
    GrB_Matrix A,                   // matrix to modify
    const GxB_Option_Field field,   // option to change
    ...                             // value to change it to
) ;
```

This usage of `GxB_set` sets the value of a matrix option, for a particular matrix. The `field` parameter can be `GxB_HYPER` or `GxB_FORMAT`.

For example, the following usage sets the hypersparsity ratio to 0.2, and the format of `GxB_BY_ROW`, for a particular matrix `A`. SuiteSparse:GraphBLAS currently applies these changes immediately, but since they are simply hints, future versions of SuiteSparse:GraphBLAS may delay the change in format if it can obtain better performance.

For performance, the matrix option should be set as soon as it is created with `GrB_Matrix_new`, so the internal transformation takes less time.

```
GxB_set (A, GxB_HYPER, 0.2) ;
GxB_set (A, GxB_FORMAT, GxB_BY_ROW) ;
```

5.5 GxB_set: set a GrB_Descriptor value

```
GrB_Info GxB_set                // set a parameter in a descriptor
(
    GrB_Descriptor desc,        // descriptor to modify
    const GrB_Desc_Field field, // parameter to change
    ...                         // value to change it to
) ;
```

This usage is identical to `GrB_Descriptor_set`, just with a name that is consistent with the other usages of this generic function. The `field` parameter can be `GrB_OUTP`, `GrB_MASK`, `GrB_INP0`, or `GrB_INP1`. Future versions will allow for different types for the third parameter. Refer to Sections 4.9.2 and 4.9.3 for details.

5.6 GxB_get: retrieve a global option

```
GrB_Info GxB_get                    // gets the current global default option
(
    const GxB_Option_Field field,   // option to query
    ...                             // return value of the global option
) ;
```

This usage of `GxB_get` retrieves the value of a global option. The `field` parameter can be `GxB_HYPER` or `GxB_FORMAT`. For example:

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

```
GxB_Format_Value s ;
GxB_get (GxB_FORMAT, &s) ;
if (s == GxB_BY_COL) printf ("all new matrices are stored by column\n") :
else printf ("all new matrices are stored by row\n") ;
```

5.7 GxB_get: retrieve a matrix option

```
GrB_Info GxB_get                    // gets the current option of a matrix
(
    const GrB_Matrix A,             // matrix to query
    const GxB_Option_Field field,   // option to query
    ...                             // return value of the matrix option
) ;
```

This usage of `GxB_get` retrieves the value of a matrix option. The `field` parameter can be `GxB_HYPER` or `GxB_FORMAT`. For example:

```
double h ;
GxB_get (A, GxB_HYPER, &h) ;
printf ("matrix A has hyper_ratio = %g\n", h) ;

GxB_Format_Value s ;
GxB_get (A, GxB_FORMAT, &s) ;
if (s == GxB_BY_COL) printf ("matrix A is stored by column\n") :
else printf ("matrix A is stored by row\n") ;
```

5.8 GxB_get: retrieve a GrB_Descriptor value

```
GrB_Info GxB_get                // get a parameter from a descriptor
(
    const GrB_Descriptor desc,  // descriptor to query; NULL means defaults
    const GrB_Desc_Field field, // parameter to query
    ...                         // value of the parameter
) ;
```

This usage is identical to `GxB_Desc_get`, just with a name that is consistent with the other usages of this generic function. The `field` parameter can be `GrB_OUTP`, `GrB_MASK`, `GrB_INP0`, `GrB_INP1` or `GxB_AxB_METHOD`. Refer to Section 4.9.4 for details.

5.9 Summary of usage of GxB_set and GxB_get

The different usages of `GxB_set` and `GxB_get` are summarized below.
To set/get the global options:

```
GxB_set (GxB_HYPER, double h) ;
GxB_set (GxB_HYPER, GxB_ALWAYS_HYPER) ;
GxB_set (GxB_HYPER, GxB_NEVER_HYPER) ;
GxB_get (GxB_HYPER, double *h) ;

GxB_set (GxB_FORMAT, GxB_BY_ROW) ;
GxB_set (GxB_FORMAT, GxB_BY_COL) ;
GxB_get (GxB_FORMAT, GxB_Format_Value *s) ;
```

To set/get a matrix option:

```
GxB_set (GrB_Matrix A, GxB_HYPER, double h) ;
GxB_set (GrB_Matrix A, GxB_HYPER, GxB_ALWAYS_HYPER) ;
GxB_set (GrB_Matrix A, GxB_HYPER, GxB_NEVER_HYPER) ;
GxB_get (GrB_Matrix A, GxB_HYPER, double *h) ;

GxB_set (GrB_Matrix A, GxB_FORMAT, GxB_BY_ROW) ;
GxB_set (GrB_Matrix A, GxB_FORMAT, GxB_BY_COL) ;
GxB_get (GrB_Matrix A, GxB_FORMAT, GxB_Format_Value *s) ;
```

To set/get a descriptor field:

```
GxB_set (GrB_Descriptor d, GrB_OUTP, GxB_DEFAULT) ;
GxB_set (GrB_Descriptor d, GrB_OUTP, GrB_REPLACE) ;

GxB_set (GrB_Descriptor d, GrB_MASK, GxB_DEFAULT) ;
GxB_set (GrB_Descriptor d, GrB_MASK, GrB_SCMP) ;

GxB_set (GrB_Descriptor d, GrB_INP0, GxB_DEFAULT) ;
GxB_set (GrB_Descriptor d, GrB_INP0, GrB_TRAN ;

GxB_set (GrB_Descriptor d, GrB_INP1, GxB_DEFAULT) ;
GxB_set (GrB_Descriptor d, GrB_INP1, GrB_TRAN);

GxB_set (GrB_Descriptor d, GxB_AxB_METHOD, GxB_DEFAULT) ;
GxB_set (GrB_Descriptor d, GxB_AxB_METHOD, GxB_AxB_GUSTAVSON) ;
GxB_set (GrB_Descriptor d, GxB_AxB_METHOD, GxB_AxB_HEAP) ;
GxB_set (GrB_Descriptor d, GxB_AxB_METHOD, GxB_AxB_DOT) ;

GxB_get (GrB_Descriptor d, GrB_Desc_Field f, GrB_Desc_Value *v) ;
```

6 SuiteSparse:GraphBLAS Colon and Index Notation

MATLAB uses a colon notation to index into matrices, such as `C=A(2:4,3:8)`, which extracts `C` as 3-by-6 submatrix from `A`, from rows 2 through 4 and columns 3 to 8 of the matrix `A`. A single colon is used to denote all rows, `C=A(:,9)`, or all columns, `C=A(12,:)`, which refers to the 9th column and 12th row of `A`, respectively. An arbitrary integer list can be given as well, such as the MATLAB statements:

```
I = [2 1 4] ;
J = [3 5] ;
C = A (I,J) ;
```

which creates the 3-by-2 matrix `C` as follows:

$$C = \begin{bmatrix} a_{2,3} & a_{2,5} \\ a_{1,3} & a_{1,5} \\ a_{4,3} & a_{4,5} \end{bmatrix}$$

The GraphBLAS API can do the equivalent of `C=A(I,J)`, `C=A(:,J)`, `C=A(I,:)`, and `C=A(:,:)`, by passing a parameter `const GrB_Index *I` as either an array of size `ni`, or as the special value `GrB_ALL`, which corresponds to the stand-alone colon `C=A(:,J)`, and the same can be done for `J`.. To compute `C=A(2:4,3:8)` in GraphBLAS requires the user application to create two explicit integer arrays `I` and `J` of size 3 and 5, respectively, and then fill them with the explicit values `[2,3,4]` and `[3,4,5,6,7,8]`. This works well if the lists are small, or if the matrix has more entries than rows or columns.

However, particularly with hypersparse matrices, the size of the explicit arrays `I` and `J` can vastly exceed the number of entries in the matrix. When using its hypersparse format, SuiteSparse:GraphBLAS allows the user application to create a `GrB_Matrix` with dimensions up to 2^{60}, with no memory constraints. The only constraint on memory usage in a hypersparse matrix is the number of entries in the matrix.

For example, creating a n-by-n matrix `A` of type `GrB_FP64` with $n = 2^{60}$ and one million entries is trivial to do in Version 2.1 (and later) of SuiteSparse:GraphBLAS, taking at most 24MB of space. SuiteSparse:GraphBLAS

Version 2.1 (or later) could do this on an old smartphone. However, using just the pure GraphBLAS API, constructing `C=A(0:(n/2),0:(n/2))` in SuiteSparse Version 2.0 would require the creation of an integer array `I` of size 2^{59}, containing the sequence 0, 1, 2, 3, ..., requiring about 4 ExaBytes of memory (4 million terabytes). This is roughly 1000 times larger than the memory size of the world's largest computer in 2018.

SuiteSparse:GraphBLAS Version 2.1 and later extends the GraphBLAS API with a full implementation of the MATLAB colon notation for integers, `I=begin:inc:end`. This extension allows the construction of the matrix `C=A(0:(n/2),0:(n/2))` in this example, with dimension 2^{59}, probably taking just milliseconds on an old smartphone.

The `GrB_extract`, `GrB_assign`, and `GrB_subassign` operations (described in the Section 7) 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.

2. To specify all rows of a matrix, use `I = GrB_ALL`. The parameter `ni` is ignored. This is equivalent to `C=A(:,J)` in MATLAB. In GraphBLAS, this is the sequence `0:(m-1)` if `A` has `m` rows, with length `|I|=m`. If `J` is used the columns of an `m`-by-`n` matrix, then `J=GrB_ALL` refers to all columns, and is the sequence `0:(n-1)`, of length `|J|=n`.

3. To specify a contiguous range of indices, such as `I=10:20` in MATLAB, the array `I` has size 2, and `ni` is passed to SuiteSparse:GraphBLAS as the special value `ni = GxB_RANGE`. The beginning index is `I[GxB_BEGIN]` and the ending index is `I[GxB_END]`. Both values must be non-negative

since `GrB_Index` is an unsigned integer (`uint64_t`). The value of `I[GxB_INC]` is ignored.

```
// to specify I = 10:20
GrB_Index I [2], ni = GxB_RANGE ;
I [GxB_BEGIN] = 10 ;      // the start of the sequence
I [GxB_END  ] = 20 ;      // the end of the sequence
```

Let $b =$ `I[GxB_BEGIN]`, let $e =$ `I[GxB_END]`, The sequence has length zero if $b > e$; otherwise the length is $|I| = (e - b) + 1$.

4. To specify a strided range of indices with a non-negative stride, such as `I=3:2:10`, the array `I` has size 3, and `ni` has the special value `GxB_STRIDE`. This is the sequence 3, 5, 7, 9, of length 4. Note that 10 does not appear in the list. The end point need not appear if the increment goes past it.

```
// to specify I = 3:2:10
GrB_Index I [3], ni = GxB_STRIDE ;
I [GxB_BEGIN ] = 3 ;      // the start of the sequence
I [GxB_INC   ] = 2 ;      // the increment
I [GxB_END   ] = 10 ;     // the end of the sequence
```

The `GxB_STRIDE` sequence is the same as the `List` generated by the following for loop:

```
int64_t k = 0 ;
GrB_Index *List = (a pointer to an array of large enough size)
for (int64_t i = I [GxB_BEGIN] ; i <= I [GxB_END] ; i += I [GxB_INC])
{
    // i is the kth entry in the sequence
    List [k++] = i ;
}
```

Then passing the explicit array `List` and its length `ni=k` has the same effect as passing in the the array `I` of size 3, with `ni=GxB_STRIDE`. The latter is simply much faster to produce, and much more efficient for SuiteSparse:GraphBLAS to process.

Let $b =$ `I[GxB_BEGIN]`, let $e =$ `I[GxB_END]`, and let $\Delta =$ `I[GxB_INC]`. The sequence has length zero if $b > e$ or $\Delta = 0$. Otherwise, the length of the sequence is

$$|I| = \left\lfloor \frac{e - b}{\Delta} \right\rfloor + 1$$

5. In MATLAB notation, if the stride is negative, the sequence is decreasing. For example, `10:-2:1` is the sequence 10, 8, 6, 4, 2, in that order. In SuiteSparse:GraphBLAS, use `ni = GxB_BACKWARDS`, with an array `I` of size 3. The following example specifies defines the equivalent of the MATLAB expression `10:-2:1` in SuiteSparse:GraphBLAS:

```
// to specify I = 10:-2:1
GrB_Index I [3], ni = GxB_BACKWARDS ;
I [GxB_BEGIN ] = 10 ;     // the start of the sequence
I [GxB_INC   ] = 2 ;      // the magnitude of the increment
I [GxB_END   ] = 1 ;      // the end of the sequence
```

   The value -2 cannot be assigned to the `GrB_Index` array `I`, since that is an unsigned type. The signed increment is represented instead with the special value `ni = GxB_BACKWARDS`. The `GxB_BACKWARDS` sequence is the same as generated by the following for loop:

```
int64_t k = 0 ;
GrB_Index *List = (a pointer to an array of large enough size)
for (int64_t i = I [GxB_BEGIN] ; i >= I [GxB_END] ; i -= I [GxB_INC])
{
    // i is the kth entry in the sequence
    List [k++] = i ;
}
```

   Let $b =$ `I[GxB_BEGIN]`, let $e =$ `I[GxB_END]`, and let $\Delta =$ `I[GxB_INC]` (note that Δ is not negative). The sequence has length zero if $b < e$ or $\Delta = 0$. Otherwise, the length of the sequence is

$$|I| = \left\lfloor \frac{b-e}{\Delta} \right\rfloor + 1$$

Since `GrB_Index` is an unsigned integer, all three values `I[GxB_BEGIN]`, `I[GxB_INC]`, and `I[GxB_END]` must be non-negative.

Just as in MATLAB, it is valid to specify an empty sequence of length zero. For example, `I = 5:3` has length zero in MATLAB and the same is true for a `GxB_RANGE` sequence in SuiteSparse:GraphBLAS, with `I[GxB_BEGIN]=5` and `I[GxB_END]=3`. This has the same effect as array `I` with `ni=0`.

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

7 GraphBLAS Operations

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

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

7.1 The GraphBLAS specification in MATLAB

SuiteSparse:GraphBLAS includes a MATLAB implementation of nearly the entire GraphBLAS specification, including all built-in types and operators. The typecasting rules and integer operator rules from GraphBLAS are implemented in MATLAB via `mexFunctions` that call the GraphBLAS routines in C. All other functions are written purely in MATLAB `M`-files, and are given names of the form `GB_spec_*`. All of these MATLAB interfaces and `M`-file functions they are provided in the software distribution of SuiteSparse:GraphBLAS. The purpose of this is two-fold:

- **Illustration and documentation:** MATLAB is so expressive, and so beautiful to read and write, that the `GB_spec_*` functions read almost like the exact specifications from the GraphBLAS API. Excerpts and condensed versions of these functions have already been used to this point in the User Guide, such as Figure 1, and the subsequent sections rely on them as well. This is why the discussion here is not just relegated to an Appendix on testing; the reader can benefit from studying the `GB_spec_*` functions to understand what a GraphBLAS operation is computing. For example, `GrB_mxm` (Section 7.2) includes a condensed and simplified version of `GB_spec_mxm`.

- **Testing:** Testing the C interface to SuiteSparse:GraphBLAS is a significant challenge since it supports so many different kinds of operations on a vast range of semirings. It is difficult to tell from looking at the result from a C function in GraphBLAS if the result is correct. Thus, each function has been written twice: once in a highly-optimized function in C, and again in a simple and elegant MATLAB function. The latter is almost a direct translation of all the mathematics behind the GraphBLAS API, so it is much easier to visually inspect the `GB_spec_*` version in MATLAB to ensure the correct mathematics are being computed.

The following functions are included in the SuiteSparse:GraphBLAS software distribution. Each has a name of the form `GB_spec_*`, and each of them is a "mimic" of a corresponding C function in GraphBLAS. Not all functions in the C API have a corresponding mimic; in particular, many of the vector functions can be computed directly with the corresponding matrix version in the MATLAB implementations. A list of these files is shown below:

MATLAB GB_spec function	corresponding GraphBLAS function or method	Section
GB_spec_accum.m	$\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$	2.3
GB_spec_mask.m	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$	2.3
GB_spec_accum_mask.m	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$	2.3
GB_spec_Vector_extractElement.m	GrB_Vector_extractElement	4.7.9
GB_spec_build.m	GrB_Matrix_build	4.8.8
GB_spec_Matrix_extractElement.m	GrB_Matrix_extractElement	4.8.10
GB_spec_extractTuples.m	GrB_Matrix_extractTuples	4.8.11
GB_spec_mxm.m	GrB_mxm	7.2
GB_spec_vxm.m	GrB_vxm	7.4
GB_spec_mxv.m	GrB_mxv	7.5
GB_spec_eWiseMult_Vector.m	GrB_eWiseMult_Vector	7.6
GB_spec_eWiseMult_Matrix.m	GrB_eWiseMult_Matrix	7.6
GB_spec_eWiseAdd_Vector.m	GrB_eWiseAdd_Vector	7.7
GB_spec_eWiseAdd_Matrix.m	GrB_eWiseAdd_Matrix	7.7
GB_spec_Vector_extract.m	GrB_Vector_extract	7.8.1
GB_spec_Matrix_extract.m	GrB_Matrix_extract	7.8.2
GB_spec_Col_extract.m	GrB_Col_extract	7.8.3
GB_spec_subassign.m	GxB_subassign	7.9
GB_spec_assign.m	GrB_assign	7.10
GB_spec_apply.m	GrB_apply	7.12
GB_spec_select.m	GxB_select	7.13
GB_spec_reduce_to_vector.m	GrB_reduce (to vector)	7.14.1
GB_spec_reduce_to_scalar.m	GrB_reduce (to scalar)	7.14.3
GB_spec_transpose.m	GrB_transpose	7.15
GB_spec_kron.m	GxB_kron	7.16

Additional files are included for creating test problems and providing inputs to the above files, or supporting functions:

MATLAB GB_spec function	purpose
GB_spec_compare.m	Compares output of C and MATLAB functions
GB_spec_random.m	Generates a random matrix
GB_spec_op.m	MATLAB mimic of built-in operators
GB_spec_operator.m	Like GrB_*Op_new
GB_spec_opsall.m	List operators, types, and semirings
GB_spec_semiring.m	Like GrB_Semiring_new
GB_spec_descriptor.m	mimics a GraphBLAS descriptor
GB_spec_identity.m	returns the identity of a monoid
GB_spec_matrix.m	conforms a MATLAB sparse matrix to GraphBLAS
GB_define.m	creates draft of GraphBLAS.h

An intensive test suite has been written that generates test graphs in MATLAB, then computes the result in both the C version of the SuiteSparse:GraphBLAS and in the MATLAB `GB_spec_*` functions. Each C function in GraphBLAS has a direct `mexFunction` interface that allow the test suite in MATLAB to call both functions.

This approach has its limitations:

- **matrix classes:** MATLAB only supports sparse double, sparse double complex, and sparse logical matrices. MATLAB can represent dense matrices in all eleven built-in GraphBLAS data types, so in all these specification `M`-files, the matrices are either in dense format in the corresponding MATLAB class, or they are held as sparse double or sparse logical, and the actual GraphBLAS type is held with it as a string member of a MATLAB `struct`. To ensure the correct typecasting is computed, most of the MATLAB scripts work on dense matrices, not sparse ones. As a result, the MATLAB `GB_spec_*` function are not meant for production use, but just for testing and illustration.

- **integer operations:** MATLAB and GraphBLAS handle integer operations differently. In MATLAB, an integer result outside the range of the integer is set to maximum or minimum integer. For example, `int8(127)+1` is `127`. This is useful for many computations such as image processing, but GraphBLAS follows the C rules instead, where integer values wrap, modulo style. For example, in GraphBLAS and in C, incrementing `(int8_t) 127` by one results in `-128`. Of course, an alternative would be for a MATLAB interface to create its own integer operators, each of which would follow the MATLAB integer rules of arithmetic. However, this would obscure the purpose of these `GB_spec_*` and `GB_mex_*` test functions, which is to test the C API of GraphBLAS. When the `GB_spec_*` functions need to perform integer computations and typecasting, they call GraphBLAS to do the work, instead doing the work in MATLAB. This ensures that the `GB_spec_*` functions obtain the same results as their GraphBLAS counterparts.

- **elegance:** to simplify testing, each MATLAB `mexFunction` interface a GraphBLAS function is a direct translation of the C API. For example, `GB_mex_mxm` is a direct interface to the GraphBLAS `GrB_mxm`, even down the order of parameters. This approach abandons some of the potential features of MATLAB for creating elegant `M`-file interfaces in a

highly usable form, such as the ability to provide fewer parameters when optional parameters are not in use. These `mexFunctions`, as written, are not meant to be usable in a user application. They are not highly documented. They are meant to be fast, and direct, to accomplish the goal of testing SuiteSparse:GraphBLAS in MATLAB and comparing their results with the corresponding `GB_spec_*` function. They are not recommended for use in general applications in MATLAB.

- **generality:** the MATLAB `mexFunction` interface needs to test the C API directly, so it must access content of SuiteSparse:GraphBLAS objects that are normally opaque to an end user application. As a result, these `mexFunctions` do not serve as a general interface to any conforming GraphBLAS implementation, but only to SuiteSparse:GraphBLAS.

In the MATLAB mimic functions, `GB_spec_*`, a GraphBLAS matrix `A` is represented as a MATLAB `struct` with the following components:

- `A.matrix`: the values of the matrix. If `A.matrix` is a sparse double matrix, it holds a typecasted copy of the values of a GraphBLAS matrix, unless the GraphBLAS matrix is also double (`GrB_FP64`).

- `A.pattern`: a logical matrix holding the pattern; `A.pattern(i,j)=true` if `(i,j)` is in the pattern of `A`, and `false` otherwise.

- `A.class`: the MATLAB class of the matrix corresponding to one of the eleven built-in types. Normally this is simply `class(A.matrix)`.

- `A.values`: most of the GraphBLAS test `mexFunctions` return their result as a MATLAB sparse matrix, in the `double` class. This works well for all types except for the 64-bit integer types, since a double has about 54 bits of mantissa which is less than the 64 bits available in a long integer. To ensure no bits are lots, these values are also returned as a vector. This enables `GB_spec_compare` to ensure the test results are identical down to the very last bit, and not just to within roundoff error. Nearly all tests, even in double precision, check for perfect equality, not just for results accurate to within round-off error.

7.2 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. The actual algorithm exploits sparsity and does not take $O(n^3)$ time, but what computes is the following:

```
[m s] = size (A.matrix) ;
[s n] = size (B.matrix) ;
T.matrix  = zeros (m, n, multiply.ztype) ;
T.pattern = zeros (m, n, 'logical') ;
T.matrix (:,:) = identity ;             % the identity of the semiring's monoid
T.class = multiply.ztype ;              % the ztype of the semiring's multiply op
A = cast (A.matrix, multiply.xtype) ;   % the xtype of the semiring's multiply op
B = cast (B.matrix, multiply.ytype) ;   % the ytype of the semiring's multiply op
for j = 1:n
    for i = 1:m
        for k = 1:s
            % T (i,j) += A (i,k) * B (k,j), using the semiring
            if (A.pattern (i,k) && B.pattern (k,j))
                z = multiply (A (i,k), B (k,j)) ;
                T.matrix  (i,j) = add (T.matrix (i,j),  z) ;
                T.pattern (i,j) = true ;
            end
        end
    end
end
```

Finally, `T` is typecasted into the type of `C`, and the results are written back into `C` via the `accum` and `Mask`, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$. The latter step is reflected in the MATLAB function `GB_spec_accum_mask.m`, discussed in Section 2.3.

Performance considerations: Suppose all matrices are in `GxB_BY_COL` format, and `B` is extremely sparse but `A` is not as sparse. Then computing `C=A*B` is very fast, and much faster then 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.

7.3 Meta-algorithm for sparse matrix multiplication

SuiteSparse:GraphBLAS includes three different methods for computing `C<M>=A*B`, and more may be added in the future. It uses a meta-algorithm to select between the methods, described below. The default behavior depends on the CSR/CSC format of each matrix (any combination can be used for the four matrices), whether or not the mask `M` is present, and the descriptor values for transposing `A` and `B`. This gives a large number of possible combinations, so the meta-algorithm that selects the method starts by reducing this space of combinations.

A matrix stored by row but not transposed has the same data structure as a matrix stored by column and transposed. To make all matrices uniform all matrices are viewed as if stored by column, by negating the transpose descriptors if they are stored by row. No data movement takes place.

```
A_transpose = true if transposed by INP0, false otherwise
B_transpose = true if transposed by INP1, false otherwise
if (A is stored by row)
    A_transpose = !A_transpose
if (B is stored by row)
    B_transpose = !B_transpose
if (C is stored by row)
    C_transpose = true
else
    C_transpose = false
if (M is present and stored by row)
```

```
    M_transpose = true
else
    M_transpose = false
```

Now all matrices are treated as if stored by column. The next phase applies the default *swap rule*, which removes the transpose of `C` via the linear algebraic property $A^{\mathsf{T}}B^{\mathsf{T}} = (BA)^{\mathsf{T}}$.

```
swap_rule = C_transpose
if (swap_rule)
    swap the matrices A and B
    A_transpose = !B_transpose
    B_transpose = !A_transpose
    C_transpose = !C_transpose
    M_transpose = !M_transpose
```

With this default swap rule, `C_transpose` is now false. All transposes below are now explicit, requiring data movement. Next, the mask is explicitly transposed.

```
if (M_transpose)
    M = M' (an explicit transpose)
    M_transpose = false
```

There are now four cases to handle:

1. If computing `C<M> = A'*B'`: the matrix `B` is explicitly tranposed and then rule (2) below is applied.

2. If computing `C<M> = A'*B`

```
if default method:
    if M is present, or if A or B have a single column, or if B or A are dense:
        use the dot method
    else
        select the heap or Gustavson method (see below)
if dot method:
    compute C<M>=A'*B
else if heap or Gustavon's method:
    A=A' ;
    compute C<M>=A*B
```

3. If computing `C<M> = A*B'`

```
if default method:
    select the heap or Gustavson method (see below)
if dot method (not selected automatically):
    A=A' ; B=B' ;
    compute C<M>=A'*B
else if heap or Gustavon's method:
    B=B' ;
    compute C<M>=A*B
```

4. If computing `C<M> = A*B`

```
if default method:
    select the heap or Gustavson method (see below)
if dot method (not selected automatically):
    A=A' ;
    compute C<M>=A'*B
else if heap or Gustavon's method:
    compute C<M>=A*B
```

The `GrB_vxm` and `GrB_mxv` operations described in Sections 7.4 and 7.5 use the same method above, where the vectors `w`, `mask`, and `u` are treated as `n`-by-1 matrices, stored by column.

The default automatic selection between the heap method or Gustavson's method for `C<M>=A*B` depends on the sparsity of `A` and `B`. Let b be the largest number of entries in any column of `B`. Let m-by-n be the dimensions of `C`. Let k be the inner dimension, so that `A` is m-by-k and `B` is k-by-n (and thus $b \leq k$).

The workspace required by the heap method is $5b$ integers, or $40b$ bytes. Gustavson's method requires $(s + 1)m$ bytes of workspace where s is the number of bytes required for the data type of `C` ($s = 8$ if `C` is `GrB_FP64`, for example). The workspace of Gustavson's method can be prohibitive if the matrices are extremely sparse since $m >> b$ or even $m >> |\mathbf{B}|$ is possible.

When the `GxB_AxB_METHOD` is `GxB_DEFAULT` and the `GxB_AxB_DOT` method has not been chosen (see above), then the following rules are used to select between the heap method and Gustavson's method.

- **Rule (1):** If $b < 2$ use the heap method. The heap will be at most 2 in size, which makes the heap method very fast. It is typically faster than Gustavson's method in this case. Example `B` matrices that fit Rule (1) include diagonal matrices, permutation matrices, or upper/lower bidiagonal matrices.

- **Rule (2):** If $|\mathbf{B}| \leq 3k$, or $|\mathbf{B}| \leq m$, or if $|\mathbf{A}| \leq \min(k, m)$ then the heap method is used if its memory requirements are much less than Gustavson's method. The heap method can be slow if b is large, however, since modifying the heap after each operation can take as much as $O(\log b)$ time. So the heap selection is penalized by a $4 \log_2 b$ factor. Thus, the heap method is used if the condition

  $$(40b)4 \log_2 b < (s + 1)m$$

  holds; otherwise Gustavson's method is used. Rules (1) and (2) mean that the heap rule is used for computing $C = PAQ$ where P and Q are permutation matrices or diagonal scaling matrices, which are common use cases. The heap method uses very little memory and is faster than Gustavson's method for these cases.

- **Rule (3):** Otherwise, Gustavson's method is used. Rules (1) and (2) ensure that the workspace requirement for Gustavson's method will not be prohibitive.

The automatic selection applies to any mixture of the format of the four matrices (CSR or CSC, and hypersparse or non-hypersparse), and any combination of the input descriptors that transpose `A` and `B`. Note that if all matrices are CSR format, `C<M>=A*B` does not explicitly transpose any matrix.

Performance considerations: With such a large space of combinations, the default meta-algorithm may not select the fastest method, which is why the method can be selected via the `GxB_AxB_METHOD` field of the descriptor.

The meta-algorithm never transposes the final result (`C=A'*B'` is never converted into `C=(B*A)'`), although that strategy can lead to better performance. A different `swap_rule` would allow for the output matrix to be transposed (with `C=(B*A)'` being computed); a user descriptor may be added in the future to allow for this selection. The user application can also do this by computing `C'=B*A`, and then transposing the result via `GrB_transpose` to obtain `C=(B*A)'=A'*B'`. This can be faster in some cases, depending on the size and sparsity of the three or four matrices (`C`, `A`, `B`, and `M` if present).

7.4 GrB_vxm: vector-matrix multiply

```
GrB_Info GrB_vxm                    // w'<Mask> = accum (w, u'*A)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_Semiring semiring,    // defines '+' and '*' for u'*A
    const GrB_Vector u,             // first input:  vector u
    const GrB_Matrix A,             // second input: matrix A
    const GrB_Descriptor desc       // descriptor for w, mask, and A
) ;
```

`GrB_vxm` multiplies a row vector `u'` times a matrix `A`. The matrix `A` may be first transposed according to `desc` (as the second input, `GrB_INP1`); the column vector `u` is never transposed via the descriptor. The inputs `u` and `A` are typecasted to match the `xtype` and `ytype` inputs, respectively, of the multiply operator of the `semiring`. Next, an intermediate column vector `t=A'*u` is computed on the `semiring` using the same method as `GrB_mxm`. Finally, the column vector `t` is typecasted from the `ztype` of the multiply operator of the `semiring` into the type of `w`, and the results are written back into `w` using the optional accumulator `accum` and `mask`.

The last step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

Performance considerations: If the `GxB_FORMAT` is `GxB_BY_COL`, `GrB_vxm` with its default descriptor can be slower than `GrB_mxv` with its default descriptor, when the vector `u` is very sparse. If the user application needs to use `GrB_vxm` repeatedly with very sparse vectors `u`, it can be faster to use a `GxB_FORMAT` of `GxB_BY_ROW`.

Using the non-default `GrB_TRAN` descriptor for `A` makes the `GrB_vxm` operation equivalent to `GrB_mxv` with its default descriptor (with the operands reversed in the multiplier, as well). The reverse is true as well; `GrB_mxv` with `GrB_TRAN` is the same as `GrB_vxm` with a default descriptor.

The breadth-first search presented in Section 8.1 of this User Guide uses `GrB_mxv` instead of `GrB_vxm`, since the graph is assumed to be symmetric and the multiplier (`AND`) is commutative, and since the default format in SuiteSparse:GraphBLAS is `GxB_BY_COL`. For the BFS algorithm in the *GraphBLAS C API Specification*, which uses `GrB_vxm`, best performance is obtained in SuiteSparse:GraphBLAS if the format of the `A` matrix is `GxB_BY_ROW`.

7.5 GrB_mxv: matrix-vector multiply

```
GrB_Info GrB_mxv                    // w<Mask> = accum (w, A*u)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GrB_Semiring semiring,    // defines '+' and '*' for A*B
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Vector u,             // second input: vector u
    const GrB_Descriptor desc       // descriptor for w, mask, and A
) ;
```

`GrB_mxv` multiplies a matrix `A` times a column vector `u`. The matrix `A` may be first transposed according to `desc` (as the first input); the column vector `u` is never transposed via the descriptor. The inputs `A` and `u` are typecasted to match the `xtype` and `ytype` inputs, respectively, of the multiply operator of the `semiring`. Next, an intermediate column vector `t=A*u` is computed on the `semiring` using the same method as `GrB_mxm`. Finally, the column vector `t` is typecasted from the `ztype` of the multiply operator of the `semiring` into the type of `w`, and the results are written back into `w` using the optional accumulator `accum` and `mask`.

The last step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

Performance considerations: Refer to the discussion of `GrB_vxm`. In SuiteSparse:GraphBLAS, `GrB_mxv` is very efficient when `u` is sparse or dense, when the default descriptor is used, and when the matrix is `GxB_BY_COL`. When `u` is very sparse and `GrB_INP0` is set to its non-default `GrB_TRAN`, then this method is not efficient if the matrix is in `GxB_BY_COL` format. If an application needs to perform `A'*u` repeatedly where `u` is very sparse, then use the `GxB_BY_ROW` format for `A` instead.

7.6 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.

7.6.1 GrB_eWiseMult_Vector: element-wise vector multiply

```
GrB_Info GrB_eWiseMult              // w<Mask> = accum (w, u.*v)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const <operator> multiply,      // defines '.*' for t=u.*v
    const GrB_Vector u,             // first input:  vector u
    const GrB_Vector v,             // second input: vector v
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GrB_eWiseMult_Vector` computes the element-wise "multiplication" of two vectors `u` and `v`, element-wise using any binary operator (not just times). The vectors are not transposed via the descriptor. The vectors `u` and `v` are first typecasted into the first and second inputs of the `multiply` operator. Next, a column vector `t` is computed, denoted $\mathbf{t} = \mathbf{u} \otimes \mathbf{v}$. The pattern of `t` is the set intersection of `u` and `v`. The result `t` has the type of the output `ztype` of the `multiply` operator.

The `operator` is typically a `GrB_BinaryOp`, but the method is type-generic for this parameter. If given a monoid (`GrB_Monoid`), the additive operator of the monoid is used as the `multiply` binary operator. If given a semiring (`GrB_Semiring`), the multiply operator of the semiring is used as the `multiply` binary operator.

The next and final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices. Note for all GraphBLAS operations, including this one, the accumulator $\mathbf{w} \odot \mathbf{t}$ is always applied in a set union manner, even though $\mathbf{t} = \mathbf{u} \otimes \mathbf{v}$ for this operation is applied in a set intersection manner.

7.6.2 GrB_eWiseMult_Matrix: element-wise matrix multiply

```
GrB_Info GrB_eWiseMult              // C<Mask> = accum (C, A.*B)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const <operator> multiply,      // defines '.*' for T=A.*B
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Matrix B,             // second input: matrix B
    const GrB_Descriptor desc       // descriptor for C, Mask, A, and B
) ;
```

`GrB_eWiseMult_Matrix` computes the element-wise "multiplication" of two matrices `A` and `B`, element-wise using any binary operator (not just times). The input matrices may be transposed first, according to the descriptor `desc`. They are then typecasted into the first and second inputs of the `multiply` operator. Next, a matrix `T` is computed, denoted $\mathbf{T} = \mathbf{A} \otimes \mathbf{B}$. The pattern of `T` is the set intersection of `A` and `B`. The result `T` has the type of the output `ztype` of the `multiply` operator.

The `multiply` operator is typically a `GrB_BinaryOp`, but the method is type-generic for this parameter. If given a monoid (`GrB_Monoid`), the additive operator of the monoid is used as the `multiply` binary operator. If given a semiring (`GrB_Semiring`), the multiply operator of the semiring is used as the `multiply` binary operator.

The operation can be expressed in MATLAB notation as:

```
[nrows, ncols] = size (A.matrix) ;
T.matrix = zeros (nrows, ncols, multiply.ztype) ;
T.class = multiply.ztype ;
p = A.pattern & B.pattern ;
A = cast (A.matrix (p), multiply.xtype) ;
B = cast (B.matrix (p), multiply.ytype) ;
T.matrix (p) = multiply (A, B) ;
T.pattern = p ;
```

The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3. Note for all GraphBLAS operations, including this one, the accumulator $\mathbf{C} \odot \mathbf{T}$ is always applied in a set union manner, even though $\mathbf{T} = \mathbf{A} \otimes \mathbf{B}$ for this operation is applied in a set intersection manner.

7.7 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.

7.7.1 GrB_eWiseAdd_Vector: element-wise vector addition

```
GrB_Info GrB_eWiseAdd               // w<Mask> = accum (w, u+v)
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const <operator> add,           // defines '+' for t=u+v
    const GrB_Vector u,             // first input:  vector u
    const GrB_Vector v,             // second input: vector v
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

GrB_eWiseAdd_Vector computes the element-wise "addition" of two vectors u and v, element-wise using any binary operator (not just plus). The vectors are not transposed via the descriptor. Entries in the intersection of u and v are first typecasted into the first and second inputs of the add operator. Next, a column vector t is computed, denoted $\mathbf{t} = \mathbf{u} \oplus \mathbf{v}$. The pattern of t is the set union of u and v. The result t has the type of the output ztype of the add operator.

The add operator is typically a GrB_BinaryOp, but the method is type-generic for this parameter. If given a monoid (GrB_Monoid), the additive operator of the monoid is used as the add binary operator. If given a semiring (GrB_Semiring), the additive operator of the monoid of the semiring is used as the add binary operator.

The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

7.7.2 GrB_eWiseAdd_Matrix: element-wise matrix addition

```
GrB_Info GrB_eWiseAdd               // C<Mask> = accum (C, A+B)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const <operator> add,           // defines '+' for T=A+B
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Matrix B,             // second input: matrix B
    const GrB_Descriptor desc       // descriptor for C, Mask, A, and B
) ;
```

`GrB_eWiseAdd_Matrix` computes the element-wise "addition" of two matrices `A` and `B`, element-wise using any binary operator (not just plus). The input matrices may be transposed first, according to the descriptor `desc`. Entries in the intersection then typecasted into the first and second inputs of the `add` operator. Next, a matrix `T` is computed, denoted $\mathbf{T} = \mathbf{A} \oplus \mathbf{B}$. The pattern of `T` is the set union of `A` and `B`. The result `T` has the type of the output `ztype` of the `add` operator.

The `add` operator is typically a `GrB_BinaryOp`, but the method is type-generic for this parameter. If given a monoid (`GrB_Monoid`), the additive operator of the monoid is used as the `add` binary operator. If given a semiring (`GrB_Semiring`), the additive operator of the monoid of the semiring is used as the `add` binary operator.

The operation can be expressed in MATLAB notation as:

```
[nrows, ncols] = size (A.matrix) ;
T.matrix = zeros (nrows, ncols, add.ztype) ;
p = A.pattern & B.pattern ;
A = GB_mex_cast (A.matrix (p), add.xtype) ;
B = GB_mex_cast (B.matrix (p), add.ytype) ;
T.matrix (p) = add (A, B) ;
p =  A.pattern & ~B.pattern ; T.matrix (p) = cast (A.matrix (p), add.ztype) ;
p = ~A.pattern &  B.pattern ; T.matrix (p) = cast (B.matrix (p), add.ztype) ;
T.pattern = A.pattern | B.pattern ;
T.class = add.ztype ;
```

Except for when typecasting is performed, this is identical to how the `accum` operator is applied in Figure 1.

The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

7.8 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.

7.8.1 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 6 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.

7.8.2 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 6 for a description of the row indices `I` and `ni`, and the column indices `J` and `nj`. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

Performance considerations: If `A` is not transposed via input descriptor: if `|I|` is small, then it is fastest if `A` is `GxB_BY_ROW`; if `|J|` is small, then it is fastest if `A` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

7.8.3 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 6 for a description of the row indices `I` and `ni`. The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

Performance considerations: If `A` is not transposed: it is fastest if the format of `A` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

7.9 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 6 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 7.10. The two operations are compared and contrasted in Section 7.11.

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

7.9.1 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 7.9.2, 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.

7.9.2 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
```

Results are not defined for any `GxB_subassign` operation if duplicate indices appear in `I` or `J`.

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GxB_BY_ROW`; if `|J|` is small, then it is fastest if the format of `C` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

7.9.3 GxB_Col_subassign: assign to a sub-column of a matrix

```
GrB_Info GxB_subassign              // C(I,j)<mask> = accum (C(I,j),u)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(I,j), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(I,j),t)
    const GrB_Vector u,             // input vector
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index j,              // column index
    const GrB_Descriptor desc       // descriptor for C(I,j) and mask
) ;
```

`GxB_Col_subassign` modifies a single sub-column of a matrix `C`. It is the same as `GxB_Matrix_subassign` where the index vector `J[0]=j` is a single column index (and thus `nj=1`), and where all matrices in `GxB_Matrix_subassign` (except `C`) consist of a single column. The `mask` vector has the same size as `u` and the sub-column `C(I,j)`. The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GxB_Matrix_subassign` for further details.

Performance considerations: `GxB_Col_subassign` is much faster than `GxB_Row_subassign` if the format of `C` is `GxB_BY_COL`. `GxB_Row_subassign` is much faster than `GxB_Col_subassign` if the format of `C` is `GxB_BY_ROW`.

7.9.4 GxB_Row_subassign: assign to a sub-row of a matrix

```
GrB_Info GxB_subassign                 // C(i,J)<mask'> = accum (C(i,J),u')
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(i,J), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(i,J),t)
    const GrB_Vector u,             // input vector
    const GrB_Index i,              // row index
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C(i,J) and mask
) ;
```

`GxB_Row_subassign` modifies a single sub-row of a matrix `C`. It is the same as `GxB_Matrix_subassign` where the index vector `I[0]=i` is a single

row index (and thus `ni=1`), and where all matrices in `GxB_Matrix_subassign` (except `C`) consist of a single row. The `mask` vector has the same size as `u` and the sub-column `C(I,j)`. The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GxB_Matrix_subassign` for further details.

Performance considerations: `GxB_Col_subassign` is much faster than `GxB_Row_subassign` if the format of `C` is `GxB_BY_COL`. `GxB_Row_subassign` is much faster than `GxB_Col_subassign` if the format of `C` is `GxB_BY_ROW`.

7.9.5 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 7.9.6.

Unlike `GrB_Vector_assign_<type>` (see Section 7.10.5), results are not defined if `I` contains duplicate indices.

7.9.6 GxB_Matrix_subassign_<type>: assign a scalar to a submatrix

```
GrB_Info GxB_subassign                  // C(I,J)<Mask> = accum (C(I,J),x)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C(I,J), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C(I,J),x)
    const <type> x,                 // scalar to assign to C(I,J)
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C(I,J) and Mask
) ;
```

`GxB_Matrix_subassign_<type>` assigns a single scalar to an entire submatrix of `C`, like the *scalar expansion* `C(I,J)=x` in MATLAB. The scalar `x` is implicitly expanded into a matrix `A` of size `ni` by `nj`, and then the matrix `A` is assigned to `C(I,J)` using the same method as in `GxB_Matrix_subassign`. Refer to that function in Section 7.9.2 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 4.8.9). Any value can be passed to this function and its type will be detected, via the `_Generic` feature of ANSI C11. For a user-defined type, `x` is a `void *` pointer that points to a memory space holding a single entry of a scalar that has exactly the same user-defined type as the matrix `C`. This user-defined type must exactly match the user-defined type of `C` since no typecasting is done between user-defined types.

If a `void *` pointer is passed in and the type of the underlying scalar does not exactly match the user-defined type of `C`, then results are undefined. No error status will be returned since GraphBLAS has no way of catching this error. Unlike `GrB_Matrix_assign_<type>` (see Section 7.10.5), results are not defined if `I` or `J` contain duplicate indices.

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GxB_BY_ROW`; if `|J|` is small, then it is fastest if the format of `C` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

7.10 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 7.9. They differ primarily in the size of the `Mask`, and how the `GrB_REPLACE` option works. Refer to Section 7.11 for a complete comparison of `GxB_subassign` and `GrB_assign`.

See Section 6 for a description of `I`, `ni`, `J`, and `nj`.

7.10.1 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 7.10.2, 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.

7.10.2 GrB_Matrix_assign: assign to a submatrix

```
GrB_Info GrB_assign                 // C<Mask>(I,J) = accum (C(I,J),A)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C(I,J),T)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index *J,             // column indices
    const GrB_Index nj,             // number of column indices
    const GrB_Descriptor desc       // descriptor for C, Mask, and A
) ;
```

`GrB_Matrix_assign` operates on a submatrix `S` of `C`, modifying it with the matrix `A`. It may also modify all of `C`, depending on the input descriptor `desc` and the `Mask`.

Step	GraphBLAS notation	description
1	$\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$	extract $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix
2	$\mathbf{S} = \mathbf{S} \odot \mathbf{A}$	apply the accumulator (but not the mask) to $\mathbf{S}$
3	$\mathbf{Z} = \mathbf{C}$	make a copy of $\mathbf{C}$
4	$\mathbf{Z}(\mathbf{I}, \mathbf{J}) = \mathbf{S}$	put the submatrix into $\mathbf{Z}(\mathbf{I}, \mathbf{J})$
5	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$	apply the mask/replace phase to all of $\mathbf{C}$

In contrast to `GxB_subassign`, the `Mask` has the same as `C`.

Step 1 extracts the submatrix and then Step 2 applies the accumulator (or $\mathbf{S} = \mathbf{A}$ if `accum` is `NULL`). The `Mask` is not yet applied.

Step 3 makes a copy of the $\mathbf{C}$ matrix, and then Step 4 writes the submatrix $\mathbf{S}$ into $\mathbf{Z}$. This is the same as Step 3 of `GxB_subassign`, except that it operates on a temporary matrix $\mathbf{Z}$.

Finally, Step 5 writes $\mathbf{Z}$ back into $\mathbf{C}$ via the `Mask`, using the Mask/Replace Phase described in Section 2.3. If `GrB_REPLACE` is enabled, then all of $\mathbf{C}$ is cleared prior to writing $\mathbf{Z}$ via the mask. As a result, the `GrB_REPLACE` option can delete entries outside the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix.

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GxB_BY_ROW`; if `|J|` is small, then it is fastest if the format of `C` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

7.10.3 GrB_Col_assign: assign to a sub-column of a matrix

```
GrB_Info GrB_assign                 // C<mask>(I,j) = accum (C(I,j),u)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Vector mask,          // optional mask for C(:,j), unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(C(I,j),t)
    const GrB_Vector u,             // input vector
    const GrB_Index *I,             // row indices
    const GrB_Index ni,             // number of row indices
    const GrB_Index j,              // column index
    const GrB_Descriptor desc       // descriptor for C(:,j) and mask
) ;
```

`GrB_Col_assign` modifies a single sub-column of a matrix `C`. It is the same as `GrB_Matrix_assign` where the index vector `J[0]=j` is a single column index, and where all matrices in `GrB_Matrix_assign` (except `C`) consist of a single column.

Unlike `GrB_Matrix_assign`, the `mask` is a vector with the same size as a single column of `C`.

The input descriptor `GrB_INP0` is ignored; the input vector `u` is not transposed. Refer to `GrB_Matrix_assign` for further details.

Performance considerations: `GrB_Col_assign` is much faster than `GrB_Row_assign` if the format of `C` is `GxB_BY_COL`. `GrB_Row_assign` is much faster than `GrB_Col_assign` if the format of `C` is `GxB_BY_ROW`.

7.10.4 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
) ;
```

`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 where all matrices in `GxB_Matrix_subassign` (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 `GxB_Matrix_subassign` for further details.

Performance considerations: `GrB_Col_assign` is much faster than `GrB_Row_assign` if the format of `C` is `GxB_BY_COL`. `GrB_Row_assign` is much faster than `GrB_Col_assign` if the format of `C` is `GxB_BY_ROW`.

7.10.5 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.

In contrast to `GxB_Vector_subassign_<type>`, results are well-defined if `I` contains duplicate indices. Duplicate indices are simply ignored.

7.10.6 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 7.10.2 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 4.8.9). Any value can be passed to this function and its type will be detected, via the `_Generic` feature of ANSI C11. For a user-defined type, `x` is a `void *` pointer that points to a memory space holding a single entry of a scalar that has exactly the same user-defined type as the matrix `C`. This user-defined type must exactly match the user-defined type of `C` since no typecasting is done between user-defined types.

If a `void *` pointer is passed in and the type of the underlying scalar does not exactly match the user-defined type of `C`, then results are undefined. No error status will be returned since GraphBLAS has no way of catching this error.

In contrast to `GxB_Matrix_subassign_<type>`, results are well-defined if `I` or `J` contain duplicate indices. Duplicate indices are simply ignored.

Performance considerations: If `A` is not transposed: if `|I|` is small, then it is fastest if the format of `C` is `GxB_BY_ROW`; if `|J|` is small, then it is fastest if the format of `C` is `GxB_BY_COL`. The opposite is true if `A` is transposed.

7.11 Comparing GrB_assign and GxB_subassign

The GxB_subassign and GrB_assign operations are very similar, but they differ in three 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_SCMP 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.

3. **Scalar expansion when duplicates appear in** I **or** J**:** They differ in how duplicate indices are treated in I and J. For both assign and subassign, results are not defined for GrB_Matrix_*assign, GrB_Vector_*assign, GrB_Row_*assign, and GrB_Col_*assign when duplicate indices appear in I or J. The scalar expansion operations, GrB_*_assign_<type>, are well-defined if duplicate indices appear (the results are the same as if duplicates are removed first from I and J). However, the scalar expansion operations GxB_*_subassign_<type> are not well-defined if duplicate indices appear in I or J.

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_SCMP`), or `I` and `J` are both `GrB_ALL`. For scalar expansion, no duplicates can appear in the index lists `I` and `J`. 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_SCMP` 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, and in how duplicate indices are treated for scalar expansion.

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 1 and 2. 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.

The first column of the table 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 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_SCMP` option.

Finally, the last column is the action taken in this case. It is left blank if no action is taken, in which case c_{ij} is not modified if present, or not inserted into $\mathbf{C}$ if not present.

repl	accum	**C**	**A**	mask	action taken by `GrB_assign` and `GxB_subassign`
-	-	c_{ij}	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, update
-	-	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
-	-	c_{ij}	-	1	delete c_{ij} because $a_{i'j'}$ not present
-	-	-	-	1	
-	-	c_{ij}	$a_{i'j'}$	0	
-	-	-	$a_{i'j'}$	0	
-	-	c_{ij}	-	0	
-	-	-	-	0	
yes	-	c_{ij}	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, update
yes	-	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
yes	-	c_{ij}	-	1	delete c_{ij} because $a_{i'j'}$ not present
yes	-	-	-	1	
yes	-	c_{ij}	$a_{i'j'}$	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	$a_{i'j'}$	0	
yes	-	c_{ij}	-	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	-	-	-	0	
-	yes	c_{ij}	$a_{i'j'}$	1	$c_{ij} = c_{ij} \odot a_{i'j'}$, apply accumulator
-	yes	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
-	yes	c_{ij}	-	1	
-	yes	-	-	1	
-	yes	c_{ij}	$a_{i'j'}$	0	
-	yes	-	$a_{i'j'}$	0	
-	yes	c_{ij}	-	0	
-	yes	-	-	0	
yes	yes	c_{ij}	$a_{i'j'}$	1	$c_{ij} = c_{ij} \odot a_{i'j'}$, apply accumulator
yes	yes	-	$a_{i'j'}$	1	$c_{ij} = a_{i'j'}$, insert
yes	yes	c_{ij}	-	1	
yes	yes	-	-	1	
yes	yes	c_{ij}	$a_{i'j'}$	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	$a_{i'j'}$	0	
yes	yes	c_{ij}	-	0	delete c_{ij} (because of `GrB_REPLACE`)
yes	yes	-	-	0	

Table 1: 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 2: Results of assign for entries outside the $\mathbf{C}(\mathbf{I}, \mathbf{J})$ submatrix. Subassign has no effect on these entries.

Table 2 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 2. 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_SCMP` is enabled via the descriptor), then the entry is deleted by `GrB_assign`.

The fourth column of Table 2 differs from Table 1, 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 1, 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 2 only applies to entries in the single row `C(i,J)` that are outside the list of indices, `J`. For `GrB_Col_assign`, Table 2 only applies to entries in the single column `C(I,j)` that are outside the list of indices, `I`.

7.11.1 Example

The difference between `GxB_subassign` and `GrB_assign` is illustrated in the following example. Consider the 2-by-2 matrix $\mathbf{C}$ where all entries are present.

$$\mathbf{C} = \begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix}$$

Suppose `GrB_REPLACE` is true, and `GrB_SCMP` is false. Let the `Mask` be:

$$\mathbf{M} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Let $\mathbf{A} = 100$, and let the index sets be $\mathbf{I} = 0$ and $\mathbf{J} = 1$. Consider the computation $\mathbf{C}\langle\mathbf{M}\rangle(0,1) = \mathbf{C}(0,1) + \mathbf{A}$, using the `GrB_assign` operation. The result is:

$$\mathbf{C} = \begin{bmatrix} 11 & 112 \\ - & 22 \end{bmatrix}.$$

The $(0,1)$ entry is updated and the $(1,0)$ entry is deleted because its `Mask` is zero. The other two entries are not modified since $\mathbf{Z} = \mathbf{C}$ outside the submatrix, and those two values are written back into $\mathbf{C}$ because their `Mask` values are 1. The $(1,0)$ entry is deleted because the entry $\mathbf{Z}(1,0) = 21$ is prevented from being written back into $\mathbf{C}$ since `Mask(1,0)=0`.

Now consider the analogous `GxB_subassign` operation. The `Mask` has the same size as $\mathbf{A}$, namely:

$$\mathbf{M} = \begin{bmatrix} 1 \end{bmatrix}.$$

After computing $\mathbf{C}(0,1)\langle\mathbf{M}\rangle = \mathbf{C}(0,1) + \mathbf{A}$, the result is

$$\mathbf{C} = \begin{bmatrix} 11 & 112 \\ 21 & 22 \end{bmatrix}.$$

Only the $\mathbf{C}(\mathbf{I},\mathbf{J})$ submatrix, the single entry $\mathbf{C}(0,1)$, is modified by `GxB_subassign`. The entry $\mathbf{C}(1,0) = 21$ is unaffected by `GxB_subassign`, but it is deleted by `GrB_assign`.

7.11.2 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 column (`GxB_BY_COL`).

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 columns (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 column of `C` with `n` entries, which can occur in practice. It is more common, however, that most columns 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 7.11. Consider assigning a large submatrix `C(I,J)=A` where `C` is the `Freescale2` matrix from the SuiteSparse Collection [DH11], of size 3 million by 3 million, with 14.3 million nonzeros. With the vectors `I=randperm(n,5500)` and `J=randperm(n,7000)` and `A` a random sparse matrix with 38,500 nonzeros, `C(I,J)=A` takes 87 seconds in MATLAB.[1] The same computation takes 0.74 seconds in SuiteSparse:GraphBLAS, a speedup of over 100. This is after finishing all pending computations in GraphBLAS and returning result to MATLAB as a valid MATLAB sparse matrix. The dominant time complexity for GraphBLAS is $O(n + c)$, where n is the dimension of `C` and c is its number of nonzeros. As a comparison, MATLAB takes just 0.42 seconds to compute `C+C'` for this matrix, which also takes time linear in the size of the matrix data structure, $O(n + c)$.

The time for submatrix assignment can be greatly reduced if `C` is hypersparse. Let $\bar{n}$ be the number of non-empty columns, where $\bar{n} < n$, and sometimes $\bar{n} << n$. The $\log n$ terms remain since they reflect the binary search inside a column, but the term n by itself is replaced by $\bar{n}$ or $\bar{n} \log \bar{n}$. The term a is replaced by $a \log \bar{n}$. The total time for a sequence of t updates to a hypersparse matrix C takes $O(t\,[a \log \bar{n} + s \log n] + (\bar{n} + c + st \log st))$ time. This assumes that `C` could include one or more dense columns, with n entries. If instead the number of entries in each column of `C` is bounded by a constant, the $\log n$ term becomes a constant. If `A` is also hypersparse, the value a is removed from the expression, replaced with its number of non-empty columns, $\bar{a}$. Suppose also t is a constant, and s is $O(\bar{a})$. Suppose the pattern of `C` changes. Suppose the number of entries in any one column of `C` is bounded by $\bar{n}$, and likewise for `A`. Then the time for the non-hypersparse case simplifies to:

$$O(a + s \log \bar{n} + (n + c + s \log s)).$$

For the hypersparse case it becomes

$$O(\bar{a} \log \bar{n} + s \log \bar{n} + (\bar{n} + c + s \log s)).$$

[1] All performance measurements in this document were done on a MacBook Pro, 2.8 GHz Intel Core i7, 16 GB Ram, OSX 10.11.6, clang 8.0.0, MATLAB R2017A.

Both of these times include the similar terms

$$O(s \log \bar{n} + (c + s \log s)),$$

which reflects the sorting of and searching of the nonzero entries themselves, in `C` and `A`. Excluding those terms and just considering the additional time, the non-hypersparse case takes an extra time of

$$O(a + n),$$

whereas the hypersparse case takes only:

$$O(\bar{a} \log \bar{n} + \bar{n}).$$

The difference is astonishing if $\bar{n} << n$ and $\bar{a} << a$. Hypersparse `C` and `A` matrices can be created with $a = n = 2^{60}$, but suppose $\bar{a} = \bar{n} = 2^{24}$, say, which is 4 million. Then $\bar{a} \log_2 \bar{n} + \bar{n}$ is only about 420 million. In stark contrast, updating the same matrices held in non-hypersparse form would take 2^{60} time and memory, a number currently beyond the reach of the world's largest supercomputer.

If the matrices `C` and `A` have no empty columns or rows, then $\bar{a} = a$ and $\bar{n} = n$. Hypersparse matrices are more costly in this case, since they still require an extra $\log n$ term to search for a specific column in `C`. As a result, SuiteSparse:GraphBLAS supports both hypersparse and non-hypersparse data structures.

7.12 GrB_apply: apply a unary operator

The `GrB_apply` function is the generic name for two specific functions: `GrB_Vector_apply` and `GrB_Matrix_apply`. 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_apply` is used.

7.12.1 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.

7.12.2 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.

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.

7.13 GxB_select: apply a select operator

The `GxB_select` function is the generic name for two specific functions: `GxB_Vector_select` and `GxB_Matrix_select`. The generic name appears in the function prototypes, but the specific function name is used when describing each variation. When discussing features that apply to both versions, the simple name `GxB_select` is used.

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

7.13.1 GxB_Vector_select: apply a select operator to a vector

```
GrB_Info GxB_select                 // w<mask> = accum (w, op(u,k))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const GxB_SelectOp op,          // operator to apply to the entries
    const GrB_Vector u,             // first input:  vector u
    const void *k,                  // optional input for the select operator
    const GrB_Descriptor desc       // descriptor for w and mask
) ;
```

`GxB_Vector_select` applies a select operator to the entries of a vector, analogous to `t = u.*op(u)` in MATLAB except the operator `op` is only applied to entries in the pattern of `u`. Implicit values outside the pattern of `u` are not affected. If the operator is not type-generic, the entries in `u` are typecasted into the `xtype` of the select operator. The vector `t` has the same type and size as `u`. The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

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

7.13.2 GxB_Matrix_select: apply a select operator to a matrix

```
GrB_Info GxB_select                 // C<Mask> = accum (C, op(A,k)) or op(A',k)
(
    GrB_Matrix C,                   // input/output matrix for results
    const GrB_Matrix Mask,          // optional mask for C, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for Z=accum(C,T)
    const GxB_SelectOp op,          // operator to apply to the entries
    const GrB_Matrix A,             // first input:  matrix A
    const void *k,                  // optional input for the select operator
    const GrB_Descriptor desc       // descriptor for C, mask, and A
) ;
```

`GxB_Matrix_select` applies a select operator to the entries of a matrix, analogous to `T = A .* op(A)` in MATLAB except the operator `op` is only applied to entries in the pattern of `A`. Implicit values outside the pattern of `A` are not affected. The input matrix `A` may be transposed first. If the operator is not type-generic, the entries in `A` are typecasted into the `xtype` of the select operator. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

The matrix `T` has the same size and type as `A` (or the transpose of `A` if the input is transposed via the descriptor). The entries of `T` are a subset of those of `A`. Each entry `A(i,j)` of `A` is passed to the `op`, as $z = f(i, j, m, n, a_{ij}, k)$, where `A` is m-by-n. If `A` is transposed first then the operator is applied to entries in the transposed matrix, `A'`. If z is returned as true, then the entry is copied into `T`, unchanged. If it returns false, the entry does not appear in `T`.

For user-defined select operators, the argument `k` is passed to the operator unchanged. For built-in operators, `k` is a pointer to an `int64_t` scalar that refers to the `k`th diagonal of the matrix. The value `k=0` specifies the main diagonal of the matrix, `k=1` is the +1 diagonal (the entries just above the main diagonal), `k=-1` is the -1 diagonal, and so on. Note that `k` must be passed as a pointer to `int64_t`, not merely as an integer. The parameter `k` is not used by `GxB_NONZERO` and may be passed as `GrB_NULL`.

The action of `GxB_select` with the built-in select operators is described in the table below. The MATLAB analogs are precise for `tril` and `triu`, but shorthand for the other operations. The MATLAB `diag` function returns a column with the diagonal, if `A` is a matrix, whereas the matrix `T` in `GxB_select` always has same size as `A` (or its transpose if the `GrB_INP0` is set to `GrB_TRAN`). In the MATLAB analog column, `diag` is as if it operates like `GxB_select`, where `T` is a matrix.

GraphBLAS name	MATLAB analog	
`GxB_TRIL`	`T=tril(A,k)`	Entries in `T` are the entries on and below the `k`th diagonal of `A`.
`GxB_TRIU`	`T=triu(A,k)`	Entries in `T` are the entries on and above the `k`th diagonal of `A`.
`GxB_DIAG`	`T=diag(A,k)`	Entries in `T` are the entries on the `k`th diagonal of `A`.
`GxB_OFFDIAG`	`T=A-diag(A,k)`	Entries in `T` are all entries not on the `k`th diagonal of `A`.
`GxB_NONZERO`	`T=A(A~=0)`	Entries in `T` are all entries in `A` that have nonzero value.

7.14 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.

7.14.1 GrB_Matrix_reduce_<op>: reduce a matrix to a vector

```
GrB_Info GrB_reduce                 // w<mask> = accum (w,reduce(A))
(
    GrB_Vector w,                   // input/output vector for results
    const GrB_Vector mask,          // optional mask for w, unused if NULL
    const GrB_BinaryOp accum,       // optional accum for z=accum(w,t)
    const <operator> reduce,        // reduce operator for t=reduce(A)
    const GrB_Matrix A,             // first input:  matrix A
    const GrB_Descriptor desc       // descriptor for w, mask, and A
) ;
```

`GrB_Matrix_reduce_<op>` is a generic name for two specific methods. Both methods reduce a matrix to a column vector using an operator, roughly analogous to `t = sum (A')` in MATLAB, in the default case, where `t` is a column vector. By default, the method reduces across the rows to obtain a column vector; use `GrB_TRAN` to reduce down the columns.

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

The input matrix `A` may be transposed first. Its entries are then typecast into the type of the `reduce` operator or monoid. The reduction is applied to all entries in `A (i,:)` to produce the scalar `t (i)`. This is done without the use of the identity value of the monoid. If the `i`th row `A (i,:)` has no entries, then `(i)` is not an entry in `t` and its value is implicit. If `A (i,:)` has a single entry, then that is the result `t (i)` and `reduce` is not applied at all for the `i`th row. Otherwise, multiple entries in row `A (i,:)` are reduced via the `reduce` operator or monoid to obtain a single scalar, the result `t (i)`.

The final step is $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{t}$, as described in Section 2.3, except that all the terms are column vectors instead of matrices.

7.14.2 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_Vector_reduce_<type>` reduces a vector to a scalar, analogous to `t = sum (u)` in MATLAB, except that in GraphBLAS any commutative and associative monoid can be used in the reduction.

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

The scalar type is any of the built-in types, or a user-defined type. In the function signature it is a C type: `bool`, `int8_t`, ... `float`, `double`, or `void *` for a user-defined type. The user-defined type must be identical to the type of the vector `u`. This cannot be checked by GraphBLAS and thus results are undefined if the types are not the same.

The descriptor is unused, but it appears in case it is needed in future versions of the GraphBLAS API. This function has no mask so its accumulator/mask step differs from the other GraphBLAS operations. It does not use the methods described in Section 2.3, but uses the following method instead.

If `accum` is `NULL`, then the scalar `t` is typecast into the type of `c`, and `c = t` is the final result. Otherwise, the scalar `t` is typecast into the `ytype` of the `accum` operator, and the value of `c` (on input) is typecast into the `xtype` of the `accum` operator. Next, the scalar `z = accum (c,t)` is computed, of the `ztype` of the `accum` operator. Finally, `z` is typecast into the final result, `c`.

Forced completion: All computations for the vector `u` are guaranteed to be finished when the method returns.

7.14.3 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_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 7.14.2.

Forced completion: All computations for the matrix `A` are guaranteed to be finished when the method returns.

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

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

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

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

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

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

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

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

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

7.16 GxB_kron: **Kronecker product**

```
GrB_Info GxB_kron                   // 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 GrB_BinaryOp 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
) ;
```

`GxB_kron` computes the Kronecker product, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathrm{kron}(\mathbf{A}, \mathbf{B})$ where

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

The $\otimes$ operator is defined by the `op` parameter. It is applied in an element-wise fashion (like `GrB_eWiseMult`), where the pattern of the submatrix $a_{ij} \otimes \mathbf{B}$ is the same as the pattern of $\mathbf{B}$ if a_{ij} is an entry in the matrix $\mathbf{A}$, or empty otherwise. The input matrices `A` and `B` can be of any dimension, and both matrices may be transposed first via the descriptor, `desc`. Entries in `A` and `B` are typecast into the input types of the `op`. The matrix `T=kron(A,B)` has the same type as the `ztype` of the binary operator, `op`. The final step is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, as described in Section 2.3.

8 Examples

Six examples on how to use GraphBLAS are described below: (1) performing a breadth-first search, (2) finding a maximal independent set, (3) creating a random matrix, (4) creating a finite-element matrix, (5) reading a matrix from a file, and (6) complex numbers as a user-defined type. The complete set of programs appears in the `Demo` directory in SuiteSparse:GraphBLAS.

8.1 Breadth-first search

The `bfs` examples in the `Demo` folder provide several examples of how to compute a breadth-first search (BFS) in GraphBLAS. The `bfs5m` function starts at a given source node `s` of an undirected graph with `n` nodes. The graph is represented as a symmetric `n`-by-`n` Boolean matrix, `A`. The matrix `A` can actually have any type; if it is not Boolean (`bool` in C, or `GrB_BOOL` in GraphBLAS), it is typecasted to Boolean by the semiring, where zero is false and nonzero is true.

The vector `v` of size `n` holds the level of each node in the BFS, where `v(i)=0` if the node has not yet been seen. This particular value makes `v` useful for another role. It can be used as a Boolean mask, since `0` is `false` and nonzero is `true`. Initially the entire `v` vector is zero.

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

Each iteration of the BFS consists of three calls to GraphBLAS. The first one uses `q` as a mask. It modifies all positions in `v` where `q` is true, setting them all to the current `level`. No accumulator or descriptor are used. Since `GrB_REPLACE` is not used and `I=GrB_ALL`, `GxB_subassign` and `GrB_assign` are identical; either can be used in this step:

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

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

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

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

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

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

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

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

Finally, a single call to GraphBLAS computes the `OR` for all entries in `q`, into a single scalar, `successor`. This value is true if `q` contains any value true, or false otherwise. If it is false, the BFS can terminate.

```
GrB_reduce (&successor, NULL, Lor, q, NULL) ;
```

The `bfs5m` function is a modified version from *The GraphBLAS C API Specification* [BMM+17b]. The method here uses `GrB_mxv` instead of `GrB_vxm`.

Another method for computing the BFS is in the `bfs6` function in the `Demo` folder. It uses `GrB_apply` and a unary operator to set the levels of the newly discovered nodes, instead of `GrB_assign`.

```
GrB_Info bfs5m              // BFS of a graph (using vector assign & reduce)
(
    GrB_Vector *v_output,   // v [i] is the BFS level of node i in the graph
    const GrB_Matrix A,     // input graph, treated as if boolean in semiring
    GrB_Index s             // starting node of the BFS
)
{
    GrB_Info info ;
    GrB_Index n ;                           // # of nodes in the graph
    GrB_Vector q = NULL ;                   // nodes visited at each level
    GrB_Vector v = NULL ;                   // result vector
    GrB_Monoid Lor = NULL ;                 // Logical-or monoid
    GrB_Semiring Boolean = NULL ;           // Boolean semiring
    GrB_Descriptor desc = NULL ;            // Descriptor for mxv
    GrB_Matrix_nrows (&n, A) ;              // n = # of rows of A
    GrB_Vector_new (&v, GrB_INT32, n) ;     // Vector<int32_t> v(n) = 0
    GrB_Vector_new (&q, GrB_BOOL, n) ;      // Vector<bool> q(n) = false
    GrB_Vector_setElement (q, true, s) ;    // q[s] = true, false elsewhere
    GrB_Monoid_new (&Lor, GrB_LOR, (bool) false) ;
    GrB_Semiring_new (&Boolean, Lor, GrB_LAND) ;
    GrB_Descriptor_new (&desc) ;
    GrB_Descriptor_set (desc, GrB_MASK, GrB_SCMP) ;     // invert the mask
    GrB_Descriptor_set (desc, GrB_OUTP, GrB_REPLACE) ;  // clear q first

    bool successor = true ; // true when some successor found
    for (int32_t level = 1 ; successor && level <= n ; level++)
    {
        // v<q> = level, using vector assign with q as the mask
        GrB_assign (v, q, NULL, level, GrB_ALL, n, NULL) ;

        // q<!v> = A ||.&& q ; finds all the unvisited successors from current
        // q, using !v as the mask
        GrB_mxv (q, v, NULL, Boolean, A, q, desc) ;

        // successor = ||(q)
        GrB_reduce (&successor, NULL, Lor, q, NULL) ;
    }
    *v_output = v ;         // return result
    GrB_free (&q) ;         // free workspace
    GrB_free (&Lor) ; GrB_free (&Boolean) ; GrB_free (&desc) ;
    return (GrB_SUCCESS) ;
}
```

8.2 Maximal independent set

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

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

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

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

```
// compute the max probability of all neighbors
GrB_mxv (neighbor_max, candidates, NULL, maxSelect2nd, A, prob, r_desc) ;
```

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

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

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

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

    while (nvals > 0)
    {
        // compute a random probability scaled by inverse of degree
        GrB_apply (prob, candidates, NULL, set_random, degrees, r_desc) ;

        // compute the max probability of all neighbors
        GrB_mxv (neighbor_max, candidates, NULL, maxSelect2nd, A, prob, r_desc) ;

        // select node if its probability is > than all its active neighbors
        GrB_eWiseAdd (new_members, NULL,NULL, GrB_GT_FP64, prob, neighbor_max,
                                                                    NULL) ;
        // add new members to independent set.
        GrB_eWiseAdd (iset, NULL, NULL, GrB_LOR, iset, new_members, NULL) ;

        // remove new members from set of candidates c = c & !new
        GrB_apply (candidates, new_members, NULL, GrB_IDENTITY_BOOL,
            candidates, sr_desc) ;

        GrB_Vector_nvals (&nvals, candidates) ;
        if (nvals == 0) { break ; }                 // early exit condition

        // Neighbors of new members can also be removed from candidates
        GrB_mxv (new_neighbors, candidates, NULL, Boolean, A,
            new_members, NULL) ;
        GrB_apply (candidates, new_neighbors, NULL, GrB_IDENTITY_BOOL,
            candidates, sr_desc) ;

        GrB_Vector_nvals (&nvals, candidates) ;
    }
```

`A` is a Boolean matrix and `prob` is a sparse real vector (of type `FP32`), where `prob(j)` is nonzero only if node `j` is a candidate. The `maxSelect2nd` semiring uses `z=SECOND(x,y)` as the multiplier operator. The row `A(i,:)` is the adjacency of node `i`, and the dot product `A(i,:)*prob` applies the `SECOND` operator on all entries that appear in the intersection of `A(i,:)` and `prob`, `z=SECOND(A(i,j),prob(j))` which is just `prob(j)` if `A(i,j)` is present. If `A(i,j)` not an explicit entry in the matrix, then this term is not computed and does not take part in the reduction by the `MAX` monoid.

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

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

All of the above work is done in a single matrix-vector multiply, with an elegant use of the `maxSelect2nd` semiring coupled with a mask. The matrix-vector multiplication is described above as if it uses dot products of rows of `A` with the column vector `prob`, but SuiteSparse:GraphBLAS does not compute it that way. Sparse dot products are much slower the optimal method for multiplying a sparse matrix times a sparse vector. The result is the same, however.

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

The resulting method is very efficient. For the `Freescale2` matrix, the algorithm finds an independent set of size 1.6 million in 1.7 seconds (on the same MacBook Pro referred to in Section 8.1), taking four iterations of the `while` loop. For comparison, removing its diagonal entries (required for the algorithm to work) takes 0.3 seconds in GraphBLAS (see Section 7.15), and simply transposing the matrix takes 0.24 seconds in both MATLAB and GraphBLAS.

8.3 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.

8.4 Creating a finite-element matrix

Suppose a finite-element matrix is being constructed, with `k=40,000` finite-element matrices, each of size `8`-by-`8`. The following operations (in pseudo-MATLAB notation) are very efficient in SuiteSparse:GraphBLAS.

```
A = sparse (m,n) ; % create an empty n-by-n sparse GraphBLAS matrix
for i = 1:k
    construct a 8-by-8 sparse or dense finite-element F
    I and J define where the matrix F is to be added:
    I = a list of 8 row indices
    J = a list of 8 column indices
    % using GrB_assign, with the 'plus' accum operator:
    A (I,J) = A (I,J) + F
end
```

If this were done in MATLAB or in GraphBLAS with blocking mode enabled, the computations would be extremely slow. This example is taken from Loren Shure's blog on MATLAB Central, *Loren on the Art of MATLAB* [Dav07], which discusses the built-in `wathen` function. In MATLAB, a far better approach is to construct a list of tuples `[I,J,X]` and to use `sparse(I,J,X,n,n)`. This is identical to creating the same list of tuples in GraphBLAS and using the `GrB_Matrix_build`, which is equally fast. The difference in time between using `sparse` or `GrB_Matrix_build`, and using submatrix assignment with blocking mode (or in MATLAB which does not have a nonblocking mode) can be extreme. For the example matrix discussed in [Dav07], using `sparse` instead of submatrix assignment in MATLAB cut the run time of `wathen` from 305 seconds down to 1.6 seconds.

In SuiteSparse:GraphBLAS, the performance of both methods is essentially identical, and roughly as fast as `sparse` in MATLAB. Inside SuiteSparse:GraphBLAS, `GrB_assign` is doing the same thing. When performing `A(I,J)=A(I,J)+F`, if it finds that it cannot quickly insert an update into the `A` matrix, it creates a list of pending tuples to be assembled later on. When the matrix is ready for use in a subsequent GraphBLAS operation (one that normally cannot use a matrix with pending computations), the tuples are assembled all at once via `GrB_Matrix_build`.

GraphBLAS operations on other matrices have no effect on when the pending updates of a matrix are completed. Thus, any GraphBLAS method or operation can be used to construct the `F` matrix in the example above, without affecting when the pending updates to `A` are completed.

The MATLAB `wathen.m` script is part of Higham's `gallery` of matrices [Hig02]. It creates a finite-element matrix with random coefficients for a 2D mesh of size `nx`-by-`ny`, a matrix formulation by Wathen [Wat87]. The pattern of the matrix is fixed; just the values are randomized. The GraphBLAS equivalent can use either `GrB_Matrix_build`, or `GrB_assign`. Both methods have good performance. The `GrB_Matrix_build` version below is about 15% to 20% faster than the MATLAB `wathen.m` function, regardless of the problem size. It uses the identical algorithm as `wathen.m`.

```
int64_t ntriplets = nx*ny*64 ;
I = malloc (ntriplets * sizeof (int64_t)) ;
J = malloc (ntriplets * sizeof (int64_t)) ;
X = malloc (ntriplets * sizeof (double )) ;
if (I == NULL || J == NULL || X == NULL)
{
    FREE_ALL ;
    return (GrB_OUT_OF_MEMORY) ;
}
ntriplets = 0 ;
for (int j = 1 ; j <= ny ; j++)
{
    for (int i = 1 ; i <= nx ; i++)
    {
        nn [0] = 3*j*nx + 2*i + 2*j + 1 ;
        nn [1] = nn [0] - 1 ;
        nn [2] = nn [1] - 1 ;
        nn [3] = (3*j-1)*nx + 2*j + i - 1 ;
        nn [4] = 3*(j-1)*nx + 2*i + 2*j - 3 ;
        nn [5] = nn [4] + 1 ;
        nn [6] = nn [5] + 1 ;
        nn [7] = nn [3] + 1 ;
        for (int krow = 0 ; krow < 8 ; krow++) nn [krow]-- ;
        for (int krow = 0 ; krow < 8 ; krow++)
        {
            for (int kcol = 0 ; kcol < 8 ; kcol++)
            {
                I [ntriplets] = nn [krow] ;
                J [ntriplets] = nn [kcol] ;
                X [ntriplets] = em (krow,kcol) ;
                ntriplets++ ;
            }
        }
    }
}
```

```
// A = sparse (I,J,X,n,n) ;
GrB_Matrix_build (A, I, J, X, ntriplets, GrB_PLUS_FP64) ;
```

The `GrB_assign` version has the advantage of not requiring the user application to construct the tuple list, and is almost as fast as using `GrB_Matrix_build`. The code is more elegant than either the MATLAB `wathen.m` function or its GraphBLAS equivalent above. Its performance is comparable with the other two methods, but slightly slower, being about 5% slower than the MATLAB `wathen`, and 20% slower than the GraphBLAS method above.

```
GrB_Matrix_new (&F, GrB_FP64, 8, 8) ;
for (int j = 1 ; j <= ny ; j++)
{
    for (int i = 1 ; i <= nx ; i++)
    {
        nn [0] = 3*j*nx + 2*i + 2*j + 1 ;
        nn [1] = nn [0] - 1 ;
        nn [2] = nn [1] - 1 ;
        nn [3] = (3*j-1)*nx + 2*j + i - 1 ;
        nn [4] = 3*(j-1)*nx + 2*i + 2*j - 3 ;
        nn [5] = nn [4] + 1 ;
        nn [6] = nn [5] + 1 ;
        nn [7] = nn [3] + 1 ;
        for (int krow = 0 ; krow < 8 ; krow++) nn [krow]-- ;
        for (int krow = 0 ; krow < 8 ; krow++)
        {
            for (int kcol = 0 ; kcol < 8 ; kcol++)
            {
                // F (krow,kcol) = em (krow, kcol)
                GrB_Matrix_setElement (F, em (krow,kcol), krow, kcol) ;
            }
        }
        // A (nn,nn) += F
        GrB_assign (A, NULL, GrB_PLUS_FP64, F, nn, 8, nn, 8, NULL) ;
    }
}
```

Since there is no `Mask`, and since `GrB_REPLACE` is not used, the call to `GrB_assign` in the example above is identical to `GxB_subassign`. Either one can be used, and their performance would be identical.

Refer to the `wathen.c` function in the `Demo` folder, which uses GraphBLAS to implement the two methods above, and two additional ones.

8.5 Reading a matrix from a file

The `read_matrix` function in the `Demo` reads in a triplet matrix from a file, one line per entry, and then uses `GrB_Matrix_build` to create the matrix. It creates a second copy with `GrB_Matrix_setElement`, just to test that method and compare the run times. A comparison of `build` versus `setElement` has already been discussed in Section 8.3.

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]` in pseudo-MATLAB notation.

If `C` is an `n`-by-`n` matrix, then `C=(C+C')/2` can be computed as follows in GraphBLAS, (the `scale2` function divides an entry by 2):

```
GrB_Descriptor_new (&dt2) ;
GrB_Descriptor_set (dt2, GrB_INP1, GrB_TRAN) ;
GrB_Matrix_new (&A, GrB_FP64, n, n) ;
GrB_eWiseAdd (A, NULL, NULL, GrB_PLUS_FP64, C, C, dt2) ;    // A=C+C'
GrB_free (&C) ;
GrB_Matrix_new (&C, GrB_FP64, n, n) ;
GrB_UnaryOp_new (&scale2_op, scale2, GrB_FP64, GrB_FP64) ;
GrB_apply (C, NULL, NULL, scale2_op, A, NULL) ;             // C=A/2
GrB_free (&A) ;
GrB_free (&scale2_op) ;
```

This is of course not nearly as elegant as `A=(A+A')/2` in MATLAB, but with minor changes it can work on any type and use any built-in operators instead of `PLUS`, or it can use any user-defined operators and types. The above code in SuiteSparse:GraphBLAS takes 0.60 seconds for the `Freescale2` matrix, slightly slower than MATLAB (0.55 seconds).

Constructing the augmented system is more complicated because GraphBLAS 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. The application must instead build a list of indices first, `I=[10, 11 ... 20]`. GraphBLAS does have a way of specifying all indices via `I=GrB_ALL`, which results in `A(:)`, but no easy way to specify a contiguous subset of indices. Thus, the following index lists `I` and `J` must first be constructed:

```
int64_t n = nrows + ncols ;
I = malloc (nrows * sizeof (int64_t)) ;
J = malloc (ncols * sizeof (int64_t)) ;
// I = 0:nrows-1
// J = nrows:n-1
if (I == NULL || J == NULL)
{
    if (I != NULL) free (I) ;
    if (J != NULL) free (J) ;
    return (GrB_OUT_OF_MEMORY) ;
}
for (int64_t k = 0 ; k < nrows ; k++) I [k] = k ;
for (int64_t k = 0 ; k < ncols ; k++) J [k] = k + nrows ;
```

Once the index lists are generated, however, the resulting GraphBLAS operations are fairly straightforward, computing `A=[0 C ; C' 0]`.

```
GrB_Descriptor_new (&dt1) ;
GrB_Descriptor_set (dt1, GrB_INP0, GrB_TRAN) ;
GrB_Matrix_new (&A, GrB_FP64, n, n) ;
// A (nrows:n-1, 0:nrows-1) = C'
GrB_assign (A, NULL, NULL, C, J, ncols, I, nrows, dt1) ;
// A (0:nrows-1, nrows:n-1) = C
GrB_assign (A, NULL, NULL, C, I, nrows, J, ncols, NULL) ;
```

This takes 1.38 seconds for the `Freescale2` matrix, almost as fast as `A=[sparse(m,m) C ; C' sparse(n,n)]` in MATLAB (1.25 seconds).

Both calls to `GrB_assign` use no accumulator, so the second one causes the partial matrix `A=[0 0 ; C' 0]` to be built first, followed by the final build of `A=[0 C ; C' 0]`. A better method, but not an obvious one, is to use the `GrB_FIRST_FP64` accumulator for both assignments. An accumulator enables SuiteSparse:GraphBLAS to determine that that entries created by the first assignment cannot be deleted by the second, and thus it need not force completion of the pending updates prior to the second assignment.

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.

8.6 Triangle counting

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

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

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

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

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

In general, the Sandia methods are the fastest of the 6 methods when implemented in GraphBLAS. The method in the KokkosKernels paper uses `(L*L).*L` via a masked matrix multiplication, but KokkosKernels stores its matrices in compressed sparse row form. SuiteSparse:GraphBLAS (by default) and MATLAB both store their matrices in compressed sparse column form, so the Sandia method is identical to `(U*U).*U` in MATLAB and $\mathbf{C}\langle\mathbf{U}\rangle = \mathbf{UU}$ in GraphBLAS. The SandiaDot and SandiaL methods do not

appear in [WDB+17], but they are named this way because they are simple extensions of the Sandia method.

The methods in MATLAB are slow because the matrix product is formed and then its entries are pruned via the element-wise multiplication (`.*`). By contrast, `GrB_mxm` only computes the entries residing in the mask, saving time and memory. This optimization is only exploited if the mask present and not complemented. Since the `minitri` method does not use a mask, its implementation in SuiteSparse:GraphBLAS has the same performance and memory requirements as the MATLAB version `nnz(A*E==2)/3`. That is, both are very slow.

Performance results are shown in the following two tables. The first table is a list of matrices from the SuiteSparse Matrix Collection [DH11], listing the matrix name, the number of rows and columns, the number of edges in the graph, and the number of triangles. The matrices were symmetrized first with `A=A+A'` and the diagonal entries were removed. The first table splits into two sets. The first set of matrices also appear in the results from the Kokkos triangles paper [WDB+17].

The next table gives performance results on these matrices, with four methods. For each method, the run time in seconds and the rate is given, where the rate is the number of edges in the graph divided by the run time (listed in millions of edges per second). The first three methods in the table are for MATLAB and the two SuiteSparse:GraphBLAS methods, on a MacBook Pro (Retina, 13inch, Late 2013), 2.8 Ghz Intel Core i7, 16 GB RAM, OSX 10.11.6, MATLAB 2017a, with the clang 8.0.0 compiler, using SuiteSparse:GraphBLAS Version 1.1.2. Only a single core was used for these results. In addition, the matrix `L=tril(A)` and/or `U=triu(A)` are used as-is without any reordering. The run times include the time to construct `L` or `U`. MATLAB failed on one matrix because `U*U` is too large. For the first set of matrices, the outer product formulation ($\mathbf{C}\langle\mathbf{U}\rangle = \mathbf{UU}$) is always faster than the dot product formulation, but this is not the case for the second set.

The last column (Kokkos) is copied directly from [WDB+17]. The Kokkos results are from their implementation of `sum(sum((L*L).*L))` using a masked sparse matrix-matrix multiply in KokkosKernels. The Kokkos results were done on an Intel Xeon Haswell (E5-2698v3, 2.3GHz), with 512 GB RAM, 32 cores and 2 hyperthreads per core, using the Intel `icc` 17.1 compiler. Unlike the other three methods, `L` is sorted by decreasing row degree, which improves the performance. The Kokkos time includes the time taken to do the sort. The run time listed is the best time obtained from several runs with 1

to 32 threads.

Comparing SuiteSparse:GraphBLAS and Kokkos is difficult since these results were obtained on different machines. Also, the results in [WDB+17] provide just the best-obtained parallel results, not the results on a single core. In addition, these results are with a reordered `L` in Kokkos, but not in SuiteSparse:GraphBLAS. Wolf et al. [WDB+17] state that reordering `L` improves the run time. However, with these many caveats, the last column lists the speedup of Kokkos over the SuiteSparse:GraphBLAS outer-product formulation. Since the Kokkos method is parallel these preliminary comparisons indicate that the sequential performance of SuiteSparse:GraphBLAS is competitive. Using up to 32 threads, Kokkos is about 3 to 18 faster than SuiteSparse:GraphBLAS, which is currently sequential (median speedup of about 9). Further comparisons are required, however. A parallel implementation of the matrix-matrix multiply in `GrB_mxm` is also in progress.

matrix	n	# edges	# triangles
SNAP/cit-HepPh	34,546	420,877	1,276,868
SNAP/cit-HepTh	27,770	352,285	1,478,735
SNAP/email-EuAll	265,214	364,481	267,313
SNAP/soc-Epinions1	75,888	405,740	1,624,481
SNAP/soc-Slashdot0811	77,360	469,180	551,724
SNAP/soc-Slashdot0902	82,168	504,230	602,592
SNAP/amazon0312	400,727	2,349,869	3,686,467
SNAP/amazon0505	410,236	2,439,437	3,951,063
SNAP/amazon0601	403,394	2,443,408	3,986,507
SNAP/cit-Patents	3,774,768	16,518,947	7,515,023
SNAP/soc-LiveJournal1	4,847,571	42,851,237	285,730,264
Gleich/wb-edu	9,845,725	46,236,105	254,718,147
SNAP/p2p-Gnutella09	8,115	26,013	2,354
Mallya/lhr71	70,304	1,492,794	160,592
Freescale/Freescale2	2,999,349	5,744,934	21,027,280
Freescale/circuit5M	5,558,326	26,983,926	31,019,473
DIMACS10/hugebubbles-00020	21,198,119	31,790,179	0
vanHeukelum/cage15	5,154,859	47,022,346	36,106,416

matrix	MATLAB		$\mathbf{C}\langle\mathbf{U}\rangle = \mathbf{L}^\intercal\mathbf{U}$		$\mathbf{C}\langle\mathbf{U}\rangle = \mathbf{UU}$		Kokkos		
	time	rate	time	rate	time	rate	time	rate	speedup
SNAP/cit-HepPh	0.363	1.16	0.180	2.47	0.049	9.59	0.0044	79.9	8.3
SNAP/cit-HepTh	0.415	0.85	0.171	2.05	0.046	8.31	0.0050	72.5	8.7
SNAP/email-EuAll	1.264	0.29	0.133	2.73	0.035	10.33	0.0058	70.7	6.8
SNAP/soc-Epinions1	0.778	0.52	0.376	1.08	0.067	6.01	0.0039	108.0	18.0
SNAP/soc-Slashdot0811	0.990	0.47	0.318	1.47	0.052	9.04	0.0061	76.8	8.5
SNAP/soc-Slashdot0902	0.985	0.51	0.339	1.49	0.059	8.61	0.0063	80.1	9.3
SNAP/amazon0312	1.285	1.83	0.514	5.32	0.306	8.61	0.0754	30.7	3.6
SNAP/amazon0505	1.018	2.07	0.545	4.48	0.297	8.21	0.0177	133.0	16.2
SNAP/amazon0601	1.018	2.40	0.563	4.34	0.296	8.27	0.0184	132.0	16.0
SNAP/cit-Patents	11.026	1.50	4.416	3.74	2.300	7.18	0.4970	31.5	4.4
SNAP/soc-LiveJournal1	11.026	0.40	39.767	1.08	10.123	4.23	0.7330	58.5	13.8
Gleich/wb-edu	67.636	0.68	8.016	5.77	3.605	12.82	0.2320	199.0	15.5
SNAP/p2p-Gnutella09	0.004	6.50	0.002	10.65	0.001	24.24			
Mallya/lhr71	0.252	5.93	0.058	25.90	0.030	50.37			
Freescale/Freescale2	0.741	7.75	0.501	11.46	0.276	20.83			
Freescale/circuit5M	mem		2.819	9.57	194.142	0.14			
DIMACS10/hugebubbles-00020	7.406	4.29	3.417	9.30	6.568	4.84			
vanHeukelum/cage15	10.187	4.62	4.407	10.67	2.443	19.25			

The outer product $\mathbf{C}\langle\mathbf{U}\rangle = \mathbf{UU}$ in SuiteSparse:GraphBLAS is very simple:

```
int64_t ntriangles ;
GrB_Index n, one = 1 ;
GrB_Matrix C, U ;
GrB_Matrix_nrows (&n, A) ;

// U = triu (A, 1)
GrB_Matrix_new (&U, GrB_UINT32, n, n) ;
GxB_select (U, NULL, NULL, GxB_TRIU, A, &one, NULL) ;

// C<U> = U*U
GrB_Matrix_new (&C, GrB_UINT32, n, n) ;
GrB_mxm (C, U, NULL, GxB_PLUS_TIMES_UINT32, U, U, NULL) ;

// ntriangles = sum (C)
GrB_reduce (&ntriangles, NULL, GxB_PLUS_INT64_MONOID, C, NULL) ;

GrB_free (&C) ;
GrB_free (&U) ;
```

The dot product method $\mathbf{C}\langle\mathbf{U}\rangle = \mathbf{L}^{\mathsf{T}}\mathbf{U}$ in SuiteSparse:GraphBLAS is similar:

```
int64_t ntriangles ;
GrB_Index n, one = 1, minusone = -1 ;
GrB_Matrix C, L, U ;
GrB_Matrix_nrows (&n, A) ;

// U = triu (A, 1)
GrB_Matrix_new (&U, GrB_UINT32, n, n) ;
GxB_select (U, NULL, NULL, GxB_TRIU, A, &one, NULL) ;

// L = tril (A,-1)
GrB_Matrix_new (&L, GrB_UINT32, n, n) ;
GxB_select (L, NULL, NULL, GxB_TRIL, A, &minusone, NULL) ;

// C<U> = L'*U
GrB_Matrix_new (&C, GrB_UINT32, n, n) ;
GrB_Descriptor_new (&d) ;
GrB_Descriptor_set (d, GrB_INP0, GrB_TRAN) ;
GrB_mxm (C, U, NULL, GxB_PLUS_TIMES_UINT32, L, U, d) ;
GrB_free (&d) ;

// ntriangles = sum (C)
GrB_reduce (&ntriangles, NULL, GxB_PLUS_INT64_MONOID, C, NULL) ;

GrB_free (&C) ;
GrB_free (&L) ;
GrB_free (&U) ;
```

For more discussion on triangle counting and k-truss algorithms, and additional results, see [Dav18b].

8.7 User-defined types and operators: double complex and struct-based

The `Demo` folder contains two working examples of user-defined types, first discussed in Section 4.1.1: `double complex`, and a user-defined `typedef` called `wildtype` with a `struct` containing a string and a 4-by-4 `float` matrix.

Double Complex: GraphBLAS does not have a native complex type, but this can be easily added as a user-defined type. The `Complex_init` function in the `usercomplex.c` file in the `Demo` folder creates the `Complex` type based on the ANSI C11 `double complex` type.

```
GrB_Type_new (&Complex, sizeof (double complex)) ;
```

Next, it creates a full suite of operators that correspond to every built-in GraphBLAS operator, both binary and unary. In addition, it creates the operators listed in the following table, where D is `double` and C is `Complex`.

name	types	MATLAB equivalent	description
`Complex_complex`	$D \times D \to C$	`z=complex(x,y)`	complex from real and imag.
`Complex_conj`	$C \to C$	`z=conj(x)`	complex conjugate
`Complex_real`	$C \to D$	`z=real(x)`	real part
`Complex_imag`	$C \to D$	`z=imag(x)`	imaginary part
`Complex_angle`	$C \to D$	`z=angle(x)`	phase angle
`Complex_complex_real`	$D \to C$	`z=complex(x,0)`	real to complex real
`Complex_complex_imag`	$D \to C$	`z=complex(0,x)`	real to complex imag.

The `Complex_init` function creates two monoids (`Complex_add_monoid` and `Complex_times_monoid`) and a semiring `Complex_plus_times` that corresponds to the conventional linear algebra for complex matrices. The include file `usercomplex.h` in the `Demo` folder is available so that this user-defined `Complex` type can easily be imported into any other user application. When the user application is done, the `Complex_finalize` function frees the `Complex` type and its operators, monoids, and semiring.

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.

9 Installing SuiteSparse:GraphBLAS

GraphBLAS makes extensive use of features in the ANSI C11 standard, and thus a C compiler supporting this version of the C standard is required. On the Mac (OS X), `clang` 8.0.0 in `Xcode` version 8.2.1 is sufficient, although earlier versions of `Xcode` may work as well. For the GNU `gcc` compiler, version 4.9 or later is required. For the Intel `icc` compiler, version 18.0 or later is required. Version 2.8.12 or later of `cmake` is required; version 3.0.0 is preferred.

To compile SuiteSparse:GraphBLAS and the demo programs, simply type `make` in the main GraphBLAS folder, which compiles the library and runs several demos.

SuiteSparse:GraphBLAS is not yet parallel, but it is thread-safe if multiple simultaneous calls are made to GraphBLAS functions. The output variables of those calls to GraphBLAS must be unique; you cannot safely modify one GraphBLAS object in parallel, with two or more simultaneous GraphBLAS functions operating on the same output object. In addition, all pending operations of objects that appear in parallel calls to GraphBLAS must be complete. This can be done for all objects via `GrB_wait`, or it can be done by calling a method or operation that forces completion of a particular object (such as `GrB_*_nvals`). If multiple parallel calls to GraphBLAS functions operate on unique inputs, then those input objects can safely have pending operations. To use GraphBLAS from parallel threads, GraphBLAS must be compiled with OpenMP so that it has access to a critical section mechanism. OpenMP is optional if the user application does not make multiple simultaneous calls to GraphBLAS.

If `cmake` or `make` fail, it might be that your default compiler does not support ANSI C11. Try another compiler. For example, try one of these options. Go into the `build` directory and type one of these:

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

By default, SuiteSparse:GraphBLAS stores its matrices by column, using the `GxB_BY_COL` format. You can change the default at compile time to `GxB_BY_ROW` using the `-DBYROW` flag when compiling. The advantage to compiling with `-DBYROW` is that a program that uses GraphBLAS with no

SuiteSparse extensions can be written that allows all matrices to be stored by row, instead of using the default format by column. For example:

```
CFLAGS=-DBYROW cmake ..
```

The user application can also use `GxB_get` to query the global option (see also Section 5.6):

```
GxB_Format_Value s ;
GxB_get (GxB_FORMAT, &s) ;
if (s == GxB_BY_COL) printf ("all new matrices are stored by column\n") :
else printf ("all new matrices are stored by row\n") ;
```

These options can also be combined. For example, to use the `gcc` compiler and to change the default format `GxB_FORMAT_DEFAULT` to `GxB_BY_ROW`, use the following `cmake` command:

```
CC=gcc CFLAGS=-DBYROW cmake ..
```

Then do `make` in the `build` directory. If this still fails, see the `CMakeLists.txt` file. You may need to pass compiler-specific options to your compiler. Locate this section in the `CMakeLists.txt` file. Use the `set` command in `cmake`, as in the example below, to set the compiler flags you need.

```
# check which compiler is being used.  If you need to make
# compiler-specific modifications, here is the place to do it.
if ("${CMAKE_C_COMPILER_ID}" STREQUAL "GNU")
    # cmake 2.8 workaround: gcc needs to be told to do ANSI C11.
    # cmake 3.0 doesn't have this problem.
    set (CMAKE_C_FLAGS  "-std=c11 -lm -fopenmp")
    ...
elseif ("${CMAKE_C_COMPILER_ID}" STREQUAL "Intel")
    ...
elseif ("${CMAKE_C_COMPILER_ID}" STREQUAL "Clang")
    ...
elseif ("${CMAKE_C_COMPILER_ID}" STREQUAL "MSVC")
    ...
endif ( )
```

Once `cmake` and `make` finish, run the demos in the `GraphBLAS/Demo` folder:

```
cd ../Demo
./demo
```

The `./demo` command is a script that runs the demos with various input matrices in the `Demo/Matrix` folder. The output of the demos will be compared with expected output files in `Demo/Output`.

To install the library in `/usr/local/lib` and `/usr/local/include`, go to the top-level GraphBLAS folder and type:

```
sudo make install
```

Several compile-time options can be selected by editing the `Source/GB.h` file, but these are meant only for code development of SuiteSparse:GraphBLAS itself, not for end-users of SuiteSparse:GraphBLAS.

To perform the extensive tests in the `Test` folder, and the statement coverage tests in `Tcov`, MATLAB R2017A is required. See the `README.txt` files in those two folders for instructions on how to run the tests.

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

NOTE: SuiteSparse:GraphBLAS has not yet been ported to Windows. However, with `cmake` the port to Windows should be straightforward (this is in progress).

10 Acknowledgments

I would like to thank Jeremy Kepner (MIT Lincoln Laboratory Supercomputing Center), and the GraphBLAS API Committee: Aydın Buluç (Lawrence Berkeley National Laboratory), Timothy G. Mattson (Intel Corporation) Scott McMillan (Software Engineering Institute at Carnegie Mellon University), José Moreira (IBM Corporation), and Carl Yang (UC Davis), for creating the GraphBLAS specification and for patiently answering my many questions while I was implementing it.

I would like to thank John Gilbert (UC Santa Barbara) for many helpful discussions on GraphBLAS, and for our decades-long conversation on sparse matrix computations, and sparse matrices in MATLAB in particular.

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

I would also 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.

References

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

[BG08] A. Buluç and J. Gilbert. On the representation and multiplication of hypersparse matrices. In *IPDPS'80: 2008 IEEE Intl. Symp. on Parallel and Distributed Processing*, pages 1–11, April 2008. https://dx.doi.org/10.1109/IPDPS.2008.4536313.

[BG12] A. Buluç and J. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012. https://dx.doi.org/10.1137/110848244.

[BMM+17a] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652, May 2017. https://dx.doi.org/10.1109/IPDPSW.2017.117.

[BMM+17b] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. The GraphBLAS C API specification. Technical report, 2017. http://graphblas.org/.

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

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

[Dav06] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, 2006.

> Provides a basic overview of many sparse matrix algorithms and a simple sparse matrix data structure. The sparse data structure used in the book is much like the one in both MATLAB and SuiteSparse:GraphBLAS. A series of 42 lectures are available on YouTube; see the link at http://faculty.cse.tamu.edu/davis/publications.html For the book, see https://dx.doi.org/10.1137/1.9780898718881

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

[Dav18a] T. A. Davis. Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the lnaguage of sparse linear algebra. *ACM Trans. on Math. Software*, (submitted), 2018. http://faculty.cse.tamu.edu/davis/publications.html.

[Dav18b] T. A. Davis. Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and K-truss. In *IEEE HPEC'18*. IEEE, 2018.

[DH11] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. on Math. Software*, 38(1):1:1–1:25, December 2011. https://dx.doi.org/10.1145/2049662.2049663.

[DRSL16] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016.

Abstract: Wilkinson defined a sparse matrix as one with enough zeros that it pays to take advantage of them. This informal yet practical definition captures the essence of the goal of direct methods for solving sparse matrix problems. They exploit the sparsity of a matrix to solve problems economically: much faster and using far less memory than if all the entries of a matrix were stored and took part in explicit computations. These methods form the backbone of a wide range of problems in computational science. A glimpse of the breadth of applications relying on sparse solvers can be seen in the origins of matrices in published matrix benchmark collections (Duff and Reid 1979a, Duff, Grimes and Lewis 1989a, Davis and Hu 2011). The goal of this survey article is to impart a working knowledge of the underlying theory and practice of sparse direct methods for solving linear systems and least-squares problems, and to provide an overview of the algorithms, data structures, and software available to solve these problems, so that the reader can both under-

stand the methods and know how best to use them. DOI: https://dx.doi.org/10.1017/S0962492916000076

[Gus78] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, 1978. https://dx.doi.org/10.1145/355791.355796.

[Hig02] N. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002. https://dx.doi.org/10.1137/1.9780898718027.

[Kep17] J. Kepner. GraphBLAS mathematics. Technical report, 2017. http://www.mit.edu/∼kepner/GraphBLAS/GraphBLAS-Math-release.pdf.

[KG11] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. SIAM, Philadelphia, PA, 2011.

From the preface: Graphs are among the most important abstract data types in computer science, and the algorithms that operate on them are critical to modern life. Graphs have been shown to be powerful tools for modeling complex problems because of their simplicity and generality. Graph algorithms are one of the pillars of mathematics, informing research in such diverse areas as combinatorial optimization, complexity theory, and topology. Algorithms on graphs are applied in many ways in today's worldfrom Web rankings to metabolic networks, from finite element meshes to semantic graphs. The current exponential growth in graph data has forced a shift to parallel computing for executing graph algorithms. Implementing parallel graph algorithms and achieving good parallel performance have proven difficult. This book addresses these challenges by exploiting the well-known duality between a canonical representation of graphs as abstract collections of vertices and edges and a sparse adjacency matrix representation. This linear algebraic approach is widely accessible to scientists and engineers who may not be formally trained in computer science. The authors show how to leverage existing parallel matrix computation techniques and the large amount of software infrastructure that exists for these computations to implement

efficient and scalable parallel graph algorithms. The benefits of this approach are reduced algorithmic complexity, ease of implementation, and improved performance. DOI: https://dx.doi.org/10.1137/1.9780898719918

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

[Wat87] A. J. Wathen. Realistic eigenvalue bounds for the Galerkin mass matrix. *IMA J. Numer. Anal.*, 7:449–457, 1987. https://dx.doi.org/10.1093/imanum/7.4.449.

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

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

Triangle counting serves as a key building block for a set of important graph algorithms in network science. In this paper, we address the IEEE HPEC Static Graph Challenge problem of triangle counting, focusing on obtaining the best parallel performance on a single multicore node. Our implementation uses a linear algebra-based approach to triangle counting that has grown out of work related to our miniTri data analytics miniapplication and our efforts to pose graph algorithms in the language of linear algebra. We leverage KokkosKernels to implement this approach efficiently on multicore architectures. Our performance results are competitive with the fastest known graph traversal-based approaches and are significantly faster than the Graph Challenge reference implementations, up to 670,000 times faster than the C++ reference and 10,000 times faster than the Python reference on a single Intel Haswell node.