GraphBLAS: graph algorithms in the language of linear algebra

Table of Contents

GraphBLAS is a library for creating graph algorithms based on sparse linear algebraic operations over semirings. Visit http://graphblas.org for more details and resources. See also the SuiteSparse:GraphBLAS User Guide in this package.

GraphBLAS: faster and more general sparse matrices for MATLAB

GraphBLAS is not only useful for creating graph algorithms; it also supports a wide range of sparse matrix data types and operations. MATLAB can compute C=A*B with just two semirings: 'plus.times.double' and 'plus.times.complex' for complex matrices. GraphBLAS has 1,040 unique built-in semirings, such as 'max.plus' (https://en.wikipedia.org/wiki/Tropical_semiring). These semirings can be used to construct a wide variety of graph algorithms, based on operations on sparse adjacency matrices.

GraphBLAS supports sparse double and single precision matrices, logical, and sparse integer matrices: int8, int16, int32, int64, uint8, uint16, uint32, and uint64. Complex matrices will be added in the future.

```
clear all
rng ('default') ;
X = 100 * rand (2) ;
G = gb (X)              % GraphBLAS copy of a matrix X, same type
```

```
G =

    2x2 GraphBLAS double matrix, standard CSC, 4 entries

    (1,1)    81.4724
    (2,1)    90.5792
    (1,2)    12.6987
    (2,2)    91.3376
```

Sparse integer matrices

Here's an int8 version of the same matrix:

```
S = int8 (G)            % convert G to a full MATLAB int8 matrix
G = gb (X, 'int8')      % a GraphBLAS sparse int8 matrix
```

```
S =

  2x2 int8 matrix

   81   12
   90   91


G =

    2x2 GraphBLAS int8_t matrix, standard CSC, 4 entries

    (1,1)   81
    (2,1)   90
    (1,2)   12
    (2,2)   91
```

Sparse single-precision matrices

Matrix operations in GraphBLAS are typically as fast, or faster than MATLAB. Here's an unfair comparison: computing X^2 with MATLAB in double precision and with GraphBLAS in single precision. You would naturally expect GraphBLAS to be faster.

Please wait ...

```
n = 1e5
X = spdiags (rand (n, 201), -100:100, n, n) ;
G = gb (X, 'single') ;
tic
G2 = G^2 ;
gb_time = toc ;
tic
X2 = X^2 ;
matlab_time = toc ;
fprintf ('\nGraphBLAS time: %g sec (in single)\n', gb_time) ;
fprintf ('MATLAB time:    %g sec (in double)\n', matlab_time) ;
fprintf ('Speedup of GraphBLAS over MATLAB: %g\n', ...
    matlab_time / gb_time) ;
```

```
n =

      100000


GraphBLAS time: 1.63696 sec (in single)
MATLAB time:    6.25715 sec (in double)
Speedup of GraphBLAS over MATLAB: 3.82242
```

Mixing MATLAB and GraphBLAS matrices

The error in the last computation is about eps('single') since GraphBLAS did its computation in single precision, while MATLAB used double precision. MATLAB and GraphBLAS matrices can be easily combined, as in X2-G2. The sparse single precision matrices take less memory space.

```
err = norm (X2 - G2, 1) / norm (X2,1)
eps ('single')
whos G G2 X X2
```

```
err =

   1.5049e-07


ans =

  single

  1.1921e-07

  Name          Size                       Bytes  Class      Attributes

  G         100000x100000            241879772  gb                   
  G2        100000x100000            481518572  gb                   
  X         100000x100000            322238408  double     sparse    
  X2        100000x100000            641756808  double     sparse    
```

Faster matrix operations

But even with standard double precision sparse matrices, GraphBLAS is typically faster than the built-in MATLAB methods. Here's a fair comparison:

```
G = gb (X) ;
tic
G2 = G^2 ;
gb_time = toc ;
err = norm (X2 - G2, 1) / norm (X2,1)
fprintf ('\nGraphBLAS time: %g sec (in double)\n', gb_time) ;
fprintf ('MATLAB time:    %g sec (in double)\n', matlab_time) ;
fprintf ('Speedup of GraphBLAS over MATLAB: %g\n', ...
    matlab_time / gb_time) ;
```

```
err =

     0


GraphBLAS time: 1.76177 sec (in double)
MATLAB time:    6.25715 sec (in double)
Speedup of GraphBLAS over MATLAB: 3.55162
```

A wide range of semirings

MATLAB can only compute C=A*B using the standard '+.*.double' and '+.*.complex' semirings. A semiring is defined in terms of a string, 'add.mult.type', where 'add' is a monoid that takes the place of the additive operator, 'mult' is the multiplicative operator, and 'type' is the data type for the two inputs to the mult operator (the type defaults to the type of A for C=A*B).

In the standard semiring, C=A*B is defined as:

```
C(i,j) = sum (A(i,:).' .* B(:,j))
```

using 'plus' as the monoid and 'times' as the multiplicative operator. But in a more general semiring, 'sum' can be any monoid, which is an associative and commutative operator that has an identity value. For example, in the 'max.plus' tropical algebra, C(i,j) for C=A*B is defined as:

```
C(i,j) = max (A(i,:).' + B(:,j))
```

This can be computed in GraphBLAS with:

```
C = gb.mxm ('max.+', A, B).
```

```
n = 3 ;
A = rand (n) ;
B = rand (n) ;
C = zeros (n) ;
for i = 1:n
    for j = 1:n
        C(i,j) = max (A (i,:).' + B (:,j)) ;
```

```
    end
end
C2 = gb.mxm ('max.+', A, B) ;
fprintf ('\nerr = norm (C-C2,1) = %g\n', norm (C-C2,1)) ;
```

```
err = norm (C-C2,1) = 0
```

The max.plus tropical semiring

Here are details of the "max.plus" tropical semiring. The identity value is -inf since max(x,-inf) = max (-inf,x) = -inf for any x.

```
gb.semiringinfo ('max.+.double') ;
```

```
    GraphBLAS Semiring: max.+.double (built-in)
    GraphBLAS Monoid: semiring->add (built-in)
    GraphBLAS BinaryOp: monoid->op (built-in) z=max(x,y)
    GraphBLAS type: ztype double size: 8
    GraphBLAS type: xtype double size: 8
    GraphBLAS type: ytype double size: 8
    identity: [    -inf ] terminal: [    inf ]

    GraphBLAS BinaryOp: semiring->multiply (built-in) z=plus(x,y)
    GraphBLAS type: ztype double size: 8
    GraphBLAS type: xtype double size: 8
    GraphBLAS type: ytype double size: 8
```

A boolean semiring

MATLAB cannot multiply two logical matrices; it converts them to double and uses the conventional +.*.double semiring instead. In GraphBLAS, this is the common Boolean 'or.and.logical' semiring, which is widely used in linear algebraic graph algorithms.

```
gb.semiringinfo ('|.&.logical') ;
```

```
    GraphBLAS Semiring: |.&.logical (built-in)
    GraphBLAS Monoid: semiring->add (built-in)
    GraphBLAS BinaryOp: monoid->op (built-in) z=or(x,y)
    GraphBLAS type: ztype bool size: 1
    GraphBLAS type: xtype bool size: 1
    GraphBLAS type: ytype bool size: 1
    identity: [   0 ] terminal: [   1 ]

    GraphBLAS BinaryOp: semiring->multiply (built-in) z=and(x,y)
    GraphBLAS type: ztype bool size: 1
    GraphBLAS type: xtype bool size: 1
    GraphBLAS type: ytype bool size: 1
```

```
clear
A = sparse (rand (3) > 0.5)
B = sparse (rand (3) > 0.2)
```

```
A =

  3x3 sparse logical array

   (2,1)      1
   (2,2)      1
   (3,2)      1
   (1,3)      1


B =

  3x3 sparse logical array

   (1,1)      1
   (2,1)      1
   (3,1)      1
   (1,2)      1
   (2,2)      1
   (3,2)      1
   (1,3)      1
   (2,3)      1
   (3,3)      1


C1 = A*B
C2 = gb (A) * gb (B)


C1 =

   (1,1)        1
   (2,1)        2
   (3,1)        1
   (1,2)        1
   (2,2)        2
   (3,2)        1
   (1,3)        1
   (2,3)        2
   (3,3)        1


C2 =

    3x3 GraphBLAS bool matrix, standard CSC, 9 entries

    (1,1)   1
    (2,1)   1
    (3,1)   1
    (1,2)   1
    (2,2)   1
    (3,2)   1
    (1,3)   1
```

```
    (2,3)    1
    (3,3)    1
```

Note that C1 is a MATLAB sparse double matrix, and contains non-binary values. C2 is a GraphBLAS logical matrix.

```
whos
gb.type (C2)
```

```
  Name      Size            Bytes  Class      Attributes

  A         3x3                68  logical    sparse
  B         3x3               113  logical    sparse
  C1        3x3               176  double     sparse
  C2        3x3              1079  gb


ans =

    'logical'
```

GraphBLAS operators, monoids, and semi-rings

The C interface for SuiteSparse:GraphBLAS allows for arbitrary types and operators to be constructed. However, the MATLAB interface to SuiteSparse:GraphBLAS is restricted to pre-defined types and operators: a mere 11 types, 66 unary operators, 275 binary operators, 44 monoids, 16 select operators, and 1,865 semirings (1,040 of which are unique, since some binary operators are equivalent: 'min.logical' and '&.logical' are the same thing, for example). The complex type and its binary operators, monoids, and semirings will be added in the near future.

That gives you a lot of tools to create all kinds of interesting graph algorithms. In this GraphBLAS/demo folder are three of them:

```
bfs_gb    % breadth-first search
dnn_gb    % sparse deep neural network (http://graphchallenge.org)
mis_gb    % maximal independent set
```

See 'help gb.binopinfo' for a list of the binary operators, and 'help gb.monoidinfo' for the ones that can be used as the additive monoid in a semiring.

```
help gb.binopinfo
```

```
 GB.BINOPINFO list the details of a GraphBLAS binary operator

  Usage

    gb.binopinfo
    gb.binopinfo (op)
    gb.binopinfo (op, type)

  For gb.binopinfo(op), the op must be a string of the form
```

'op.type', where 'op' is listed below. The second usage allows the type to be omitted from the first argument, as just 'op'. This is valid for all GraphBLAS operations, since the type defaults to the type of the input matrices. However, gb.binopinfo does not have a default type and thus one must be provided, either in the op as gb.binopinfo ('+.double'), or in the second argument, gb.binopinfo ('+', 'double').

*The MATLAB interface to GraphBLAS provides for 25 different binary operators, each of which may be used with any of the 11 types, for a total of 25*11 = 275 valid binary operators. Binary operators are defined by a string of the form 'op.type', or just 'op'. In the latter case, the type defaults to the type of the matrix inputs to the GraphBLAS operation.*

The 6 comparator operators come in two flavors. For the is operators, the result has the same type as the inputs, x and y, with 1 for true and 0 for false. For example isgt.double (pi, 3.0) is the double value 1.0. For the second set of 6 operators (eq, ne, gt, lt, ge, le), the result is always logical (true or false). In a semiring, the type of the add monoid must exactly match the type of the output of the multiply operator, and thus 'plus.iseq.double' is valid (counting how many terms are equal). The 'plus.eq.double' semiring is valid, but not the same semiring since the 'plus' of 'plus.eq.double' has a logical type and is thus equivalent to 'or.eq.double'. The 'or.eq' is true if any terms are equal and false otherwise (it does not count the number of terms that are equal).*

The following binary operators are available. Many have equivalent synonyms, so that '1st' and 'first' both define the first(x,y) = x operator.

```
operator name(s) f(x,y)         |   operator names(s) f(x,y)
---------------- ------         |   ----------------- ------
1st first        x              |   iseq              x == y
2nd second       y              |   isne              x ~= y
min              min(x,y)       |   isgt              x > y
max              max(x,y)       |   islt              x < y
+   plus         x+y            |   isge              x >= y
-   minus        x-y            |   isle              x <= y
rminus           y-x            |   ==   eq           x == y
*   times        x*y            |   ~=   ne           x ~= y
/   div          x/y            |   >    gt           x > y
\   rdiv         y/x            |   <    lt           x < y
|   || or  lor   x | y          |   >=   ge           x >= y
&   && and land  x & y          |   <=   le           x <= y
xor lxor         xor(x,y)       |
```

The three logical operators, lor, land, and lxor, also come in 11 types. z = lor.double (x,y) tests the condition (x~=0) || (y~=0), and returns the double value 1.0 if true, or 0.0 if false.

Example:

```
    % valid binary operators
    gb.binopinfo ('+.double') ;
    gb.binopinfo ('1st.int32') ;

    % invalid binary operator (an error; this is a unary op):
    gb.binopinfo ('abs.double') ;

  gb.binopinfo generates an error for an invalid op, so user code can
  test the validity of an op with the MATLAB try/catch mechanism.

  See also gb, gb.unopinfo, gb.semiringinfo, gb.descriptorinfo.
```

```
help gb.monoidinfo
```

```
 GB.MONOIDINFO list the details of a GraphBLAS monoid

  Usage

    gb.monoidinfo
    gb.monoidinfo (monoid)
    gb.monoidinfo (monoid, type)

  For gb.monoidinfo(op), the op must be a string of the form
  'op.type', where 'op' is listed below.  The second usage allows the
  type to be omitted from the first argument, as just 'op'.  This is
  valid for all GraphBLAS operations, since the type defaults to the
  type of the input matrices.  However, gb.monoidinfo does not have a
  default type and thus one must be provided, either in the op as
  gb.monoidinfo ('+.double'), or in the second argument,
  gb.monoidinfo ('+', 'double').

  The MATLAB interface to GraphBLAS provides for 44 different
  monoids.  The valid monoids are: '+', '*', 'max', and 'min' for all
  but the 'logical' type, and '|', '&', 'xor', and 'ne' for the
  'logical' type.

  Example:

    % valid monoids
    gb.monoidinfo ('+.double') ;
    gb.monoidinfo ('*.int32') ;

    % invalid monoids
    gb.monoidinfo ('1st.int32') ;
    gb.monoidinfo ('abs.double') ;

  gb.monoidinfo generates an error for an invalid monoid, so user
  code can test the validity of an op with the MATLAB try/catch
  mechanism.

  See also gb.unopinfo, gb.binopinfo, gb.semiringinfo,
  gb.descriptorinfo.
```

Element-wise operations

Binary operators can be used in element-wise matrix operations, like C=A+B and C=A.*B. For the matrix addition C=A+B, the pattern of C is the set union of A and B, and the '+' operator is applied for entries in the intersection. Entries in A but not B, or in B but not A, are assigned to C without using the operator. The '+' operator is used for C=A+B but any operator can be used with gb.eadd.

```
A = gb (sprand (3, 3, 0.5)) ;
B = gb (sprand (3, 3, 0.5)) ;
C1 = A + B
C2 = gb.eadd ('+', A, B)
C1-C2
```

```
C1 =

    3x3 GraphBLAS double matrix, standard CSC, 7 entries

    (1,1)    0.666139
    (3,1)    0.735859
    (1,2)    1.47841
    (2,2)    0.146938
    (3,2)    0.566879
    (2,3)    0.248635
    (3,3)    0.104226


C2 =

    3x3 GraphBLAS double matrix, standard CSC, 7 entries

    (1,1)    0.666139
    (3,1)    0.735859
    (1,2)    1.47841
    (2,2)    0.146938
    (3,2)    0.566879
    (2,3)    0.248635
    (3,3)    0.104226


ans =

    3x3 GraphBLAS double matrix, standard CSC, 7 entries

    (1,1)    0
    (3,1)    0
    (1,2)    0
    (2,2)    0
    (3,2)    0
    (2,3)    0
    (3,3)    0
```

Subtracting two matrices

A-B and gb.eadd ('-', A, B) are not the same thing, since the '-' operator is not applied to an entry that is in B but not A.

```
C1 = A-B
C2 = gb.eadd ('-', A, B)
```

```
C1 =

    3x3 GraphBLAS double matrix, standard CSC, 7 entries

    (1,1)    -0.666139
    (3,1)    -0.735859
    (1,2)    -0.334348
    (2,2)    -0.146938
    (3,2)    0.566879
    (2,3)    0.248635
    (3,3)    0.104226


C2 =

    3x3 GraphBLAS double matrix, standard CSC, 7 entries

    (1,1)    0.666139
    (3,1)    0.735859
    (1,2)    -0.334348
    (2,2)    0.146938
    (3,2)    0.566879
    (2,3)    0.248635
    (3,3)    0.104226
```

But these give the same result

```
C1 = A-B
C2 = gb.eadd ('+', A, gb.apply ('-', B))
C1-C2
```

```
C1 =

    3x3 GraphBLAS double matrix, standard CSC, 7 entries

    (1,1)    -0.666139
    (3,1)    -0.735859
    (1,2)    -0.334348
    (2,2)    -0.146938
    (3,2)    0.566879
    (2,3)    0.248635
    (3,3)    0.104226
```

```
C2 =

    3x3 GraphBLAS double matrix, standard CSC, 7 entries

    (1,1)    -0.666139
    (3,1)    -0.735859
    (1,2)    -0.334348
    (2,2)    -0.146938
    (3,2)    0.566879
    (2,3)    0.248635
    (3,3)    0.104226


ans =

    3x3 GraphBLAS double matrix, standard CSC, 7 entries

    (1,1)    0
    (3,1)    0
    (1,2)    0
    (2,2)    0
    (3,2)    0
    (2,3)    0
    (3,3)    0
```

Element-wise 'multiplication'

For C = A.*B, the result C is the set intersection of the pattern of A and B. The operator is applied to entries in both A and B. Entries in A but not B, or B but not A, do not appear in the result C.

```
C1 = A.*B
C2 = gb.emult ('*', A, B)
C3 = double (A) .* double (B)
```

```
C1 =

    3x3 GraphBLAS double matrix, standard CSC, 1 entries

    (1,2)    0.518474


C2 =

    3x3 GraphBLAS double matrix, standard CSC, 1 entries

    (1,2)    0.518474


C3 =

   (1,2)        0.5185
```

Just as in gb.eadd, any operator can be used in gb.emult:

```
A
B
C2 = gb.emult ('max', A, B)


A =

    3x3 GraphBLAS double matrix, standard CSC, 4 entries

    (1,2)    0.572029
    (3,2)    0.566879
    (2,3)    0.248635
    (3,3)    0.104226


B =

    3x3 GraphBLAS double matrix, standard CSC, 4 entries

    (1,1)    0.666139
    (3,1)    0.735859
    (1,2)    0.906378
    (2,2)    0.146938


C2 =

    3x3 GraphBLAS double matrix, standard CSC, 1 entries

    (1,2)    0.906378
```

Overloaded operators

The following operators all work as you would expect for any matrix. The matrices A and B can be GraphBLAS matrices, or MATLAB sparse or dense matrices, in any combination, or scalars where appropriate:

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

For A^b, b must be a non-negative integer.

```
A
B
C1 = [A B]
C2 = [double(A) double(B)] ;
assert (isequal (double (C1), C2))
C1 = A^2
C2 = double (A)^2 ;
assert (isequal (double (C1), C2))
```

```
C1 = A (1:2,2:end)
A = double (A) ;
C2 = A (1:2,2:end) ;
assert (isequal (double (C1), C2))


A =

    3x3 GraphBLAS double matrix, standard CSC, 4 entries

    (1,2)    0.572029
    (3,2)    0.566879
    (2,3)    0.248635
    (3,3)    0.104226


B =

    3x3 GraphBLAS double matrix, standard CSC, 4 entries

    (1,1)    0.666139
    (3,1)    0.735859
    (1,2)    0.906378
    (2,2)    0.146938


C1 =

    3x6 GraphBLAS double matrix, standard CSC, 8 entries

    (1,2)    0.572029
    (3,2)    0.566879
    (2,3)    0.248635
    (3,3)    0.104226
    (1,4)    0.666139
    (3,4)    0.735859
    (1,5)    0.906378
    (2,5)    0.146938


C1 =

    3x3 GraphBLAS double matrix, standard CSC, 5 entries

    (2,2)    0.140946
    (3,2)    0.0590838
    (1,3)    0.142227
    (2,3)    0.0259144
    (3,3)    0.151809


C1 =

    2x2 GraphBLAS double matrix, standard CSC, 2 entries
```

```
   (1,1)     0.572029
   (2,2)     0.248635
```

Overloaded functions

Many MATLAB built-in functions can be used with GraphBLAS matrices:

A few differences with the built-in functions:

```
S = sparse (G)      % makes a copy of a gb matrix
F = full (G)        % adds explicit zeros, so numel(F)==nnz(F)
F = full (G,id)     % adds explicit identity values to a gb matrix
disp (G, level)     % display a gb matrix G; level=2 is the default.
e = nnz (G)         % # of entries in a gb matrix G; some can be zero
X = nonzeros (G)    % all the entries of G; some can be zero
```

In the list below, the first set of Methods are overloaded built-in methods. They are used as-is on GraphBLAS matrices, such as C=abs(G). The Static methods are prefixed with "gb.", as in C = gb.apply (...).

```
methods gb
```

```
Methods for class gb:

abs               horzcat           le                single
all               int16             length            size
amd               int32             logical           sparse
and               int64             lt                spfun
any               int8              max               spones
bandwidth         isa               min               sqrt
ceil              isbanded          minus             subsasgn
colamd            isdiag            mldivide          subsindex
complex           isempty           mpower            subsref
conj              isfinite          mrdivide          sum
ctranspose        isfloat           mtimes            symamd
diag              ishermitian       ne                symrcm
disp              isinf             nnz               times
display           isinteger         nonzeros          transpose
dmperm            islogical         norm              tril
double            ismatrix          not               triu
eig               isnan             numel             uint16
end               isnumeric         nzmax             uint32
eps               isreal            or                uint64
eq                isscalar          plus              uint8
find              issparse          power             uminus
fix               issymmetric       prod              uplus
floor             istril            rdivide           vertcat
full              istriu            real
gb                isvector          repmat
ge                kron              round
gt                ldivide           sign
```

```
Static methods:

apply           empty           gbtranspose     subassign
assign          emult           monoidinfo      threads
binopinfo       expand          mxm             type
build           extract         nvals           unopinfo
chunk           extracttuples   reduce          vreduce
clear           eye             select
descriptorinfo  format          semiringinfo
eadd            gbkron          speye
```

Zeros are handled differently

Explicit zeros cannot be automatically dropped from a GraphBLAS matrix, like they are in MATLAB sparse matrices. In a shortest-path problem, for example, an edge A(i,j) that is missing has an infinite weight, (the monoid identity of min(x,y) is +inf). A zero edge weight A(i,j)=0 is very different from an entry that is not present in A. However, if a GraphBLAS matrix is converted into a MATLAB sparse matrix, explicit zeros are dropped, which is the convention for a MATLAB sparse matrix. They can also be dropped from a GraphBLAS matrix using the gb.select method.

```
G = gb (magic (3)) ;
G (1,1) = 0      % G(1,1) still appears as an explicit entry
A = double (G)   % but it's dropped when converted to MATLAB sparse
H = gb.select ('nonzero', G)  % drops the explicit zeros from G
fprintf ('nnz (G): %d  nnz (A): %g nnz (H): %g\n', ...
    nnz (G), nnz (A), nnz (H)) ;
```

```
G =

    3x3 GraphBLAS double matrix, standard CSC, 9 entries

    (1,1)    0
    (2,1)    3
    (3,1)    4
    (1,2)    1
    (2,2)    5
    (3,2)    9
    (1,3)    6
    (2,3)    7
    (3,3)    2


A =

   (2,1)        3
   (3,1)        4
   (1,2)        1
   (2,2)        5
   (3,2)        9
   (1,3)        6
   (2,3)        7
   (3,3)        2
```

```
H =

    3x3 GraphBLAS double matrix, standard CSC, 8 entries

    (2,1)    3
    (3,1)    4
    (1,2)    1
    (2,2)    5
    (3,2)    9
    (1,3)    6
    (2,3)    7
    (3,3)    2

nnz (G): 9  nnz (A): 8 nnz (H): 8
```

Displaying contents of a GraphBLAS matrix

Unlike MATLAB, the default is to display just a few entries of a gb matrix. Here are all 100 entries of a 10-by-10 matrix, using a non-default disp(G,3):

```
G = gb (rand (10)) ;
% display everything:
disp (G,3)
```

```
G =

    10x10 GraphBLAS double matrix, standard CSC, 100 entries

    (1,1)    0.0342763
    (2,1)    0.17802
    (3,1)    0.887592
    (4,1)    0.889828
    (5,1)    0.769149
    (6,1)    0.00497062
    (7,1)    0.735693
    (8,1)    0.488349
    (9,1)    0.332817
    (10,1)    0.0273313
    (1,2)    0.467212
    (2,2)    0.796714
    (3,2)    0.849463
    (4,2)    0.965361
    (5,2)    0.902248
    (6,2)    0.0363252
    (7,2)    0.708068
    (8,2)    0.322919
    (9,2)    0.700716
    (10,2)    0.472957
    (1,3)    0.204363
    (2,3)    0.00931977
    (3,3)    0.565881
```

```
(4,3)    0.183435
(5,3)    0.00843818
(6,3)    0.284938
(7,3)    0.706156
(8,3)    0.909475
(9,3)    0.84868
(10,3)    0.564605
(1,4)    0.075183
(2,4)    0.535293
(3,4)    0.072324
(4,4)    0.515373
(5,4)    0.926149
(6,4)    0.949252
(7,4)    0.0478888
(8,4)    0.523767
(9,4)    0.167203
(10,4)    0.28341
(1,5)    0.122669
(2,5)    0.441267
(3,5)    0.157113
(4,5)    0.302479
(5,5)    0.758486
(6,5)    0.910563
(7,5)    0.0246916
(8,5)    0.232421
(9,5)    0.38018
(10,5)    0.677531
(1,6)    0.869074
(2,6)    0.471459
(3,6)    0.624929
(4,6)    0.987186
(5,6)    0.282885
(6,6)    0.843833
(7,6)    0.869597
(8,6)    0.308209
(9,6)    0.201332
(10,6)    0.706603
(1,7)    0.563222
(2,7)    0.575795
(3,7)    0.056376
(4,7)    0.73412
(5,7)    0.608022
(6,7)    0.0400164
(7,7)    0.540801
(8,7)    0.023064
(9,7)    0.165682
(10,7)    0.250393
(1,8)    0.23865
(2,8)    0.232033
(3,8)    0.303191
(4,8)    0.579934
(5,8)    0.267751
(6,8)    0.916376
(7,8)    0.833499
```

```
    (8,8)      0.978692
    (9,8)      0.734445
    (10,8)     0.102896
    (1,9)      0.353059
    (2,9)      0.738955
    (3,9)      0.57539
    (4,9)      0.751433
    (5,9)      0.93256
    (6,9)      0.281622
    (7,9)      0.51302
    (8,9)      0.24406
    (9,9)      0.950086
    (10,9)     0.303638
    (1,10)     0.563593
    (2,10)     0.705101
    (3,10)     0.0604146
    (4,10)     0.672065
    (5,10)     0.359793
    (6,10)     0.62931
    (7,10)     0.977758
    (8,10)     0.394328
    (9,10)     0.765651
    (10,10)    0.457809
```

That was disp(G,3), so every entry was printed. It's a little long, so the default is not to print everything.

With the default display (level = 2):

```
G
```

```
G =

    10x10 GraphBLAS double matrix, standard CSC, 100 entries

    (1,1)      0.0342763
    (2,1)      0.17802
    (3,1)      0.887592
    (4,1)      0.889828
    (5,1)      0.769149
    (6,1)      0.00497062
    (7,1)      0.735693
    (8,1)      0.488349
    (9,1)      0.332817
    (10,1)     0.0273313
    (1,2)      0.467212
    (2,2)      0.796714
    (3,2)      0.849463
    (4,2)      0.965361
    (5,2)      0.902248
    (6,2)      0.0363252
    (7,2)      0.708068
    (8,2)      0.322919
```

```
    (9,2)    0.700716
    (10,2)    0.472957
    (1,3)    0.204363
    (2,3)    0.00931977
    (3,3)    0.565881
    (4,3)    0.183435
    (5,3)    0.00843818
    (6,3)    0.284938
    (7,3)    0.706156
    (8,3)    0.909475
    (9,3)    0.84868
    (10,3)    0.564605
    ...
```

That was disp(G,2) or just display(G), which is what is printed by a MATLAB statement that doesn't have a trailing semicolon. With level = 1, disp(G,1) gives just a terse summary:

```
disp (G,1)
```

```
G =

    10x10 GraphBLAS double matrix, standard CSC, 100 entries
```

Storing a matrix by row or by column

MATLAB stores its sparse matrices by column, refered to as 'standard CSC' in SuiteSparse:GraphBLAS. In the CSC (compressed sparse column) format, each column of the matrix is stored as a list of entries, with their value and row index. In the CSR (compressed sparse row) format, each row is stored as a list of values and their column indices. GraphBLAS uses both CSC and CSR, and the two formats can be intermixed arbitrarily. In its C interface, the default format is CSR. However, for better compatibility with MATLAB, this MATLAB interface for SuiteSparse:GraphBLAS uses CSC by default instead.

```
rng ('default') ;
gb.clear ;                          % clear all prior GraphBLAS settings
default_format_is = gb.format
C = sparse (rand (2))
G = gb (C)
gb.format (G)
```

```
default_format_is =

    'by col'


C =

   (1,1)       0.8147
   (2,1)       0.9058
   (1,2)       0.1270
```

```
   (2,2)       0.9134


G =

    2x2 GraphBLAS double matrix, standard CSC, 4 entries

    (1,1)    0.814724
    (2,1)    0.905792
    (1,2)    0.126987
    (2,2)    0.913376


ans =

    'by col'
```

Many graph algorithms work better in CSR format, with matrices stored by row. For example, it is common to use A(i,j) for the edge (i,j), and many graph algorithms need to access the out-adjacencies of nodes, which is the row A(i,:) for node i. If the CSR format is desired, gb.format ('by row') tells GraphBLAS to create all subsequent matrices in the CSR format. Converting from a MATLAB sparse matrix (in standard CSC format) takes a little more time (requiring a transpose), but subsequent graph algorithms can be faster.

```
gb.format ('by row') ;
default_format_is = gb.format
G = gb (C)
The_format_for_G_is = gb.format (G)
default_format_is_now_back_to = gb.format ('by col')
H = gb (C)
The_format_for_H_is = gb.format (H)
But_G_is_still = gb.format (G)
err = norm (H-G,1)
```

```
default_format_is =

    'by col'


G =

    2x2 GraphBLAS double matrix, standard CSR, 4 entries

    (1,1)    0.814724
    (1,2)    0.126987
    (2,1)    0.905792
    (2,2)    0.913376


The_format_for_G_is =

    'by row'
```

```
default_format_is_now_back_to =

    'by col'


H =

    2x2 GraphBLAS double matrix, standard CSC, 4 entries

    (1,1)    0.814724
    (2,1)    0.905792
    (1,2)    0.126987
    (2,2)    0.913376


The_format_for_H_is =

    'by col'


But_G_is_still =

    'by row'


err =

     0
```

Hypersparse matrices

SuiteSparse:GraphBLAS can use two kinds of sparse matrix data structures: standard and hypersparse, for both CSC and CSR formats. In the standard CSC format used in MATLAB, an m-by-n matrix A takes O(n +nnz(A)) space. MATLAB can create huge column vectors, but not huge matrices (when n is huge).

```
clear all
[c, huge] = computer ;
C = sparse (huge, 1)    % MATLAB can create a huge-by-1 sparse column
try
    C = sparse (huge, huge)     % but this fails
catch me
    error_expected = me
end


C =

   All zero sparse: 281474976710655x1


error_expected =

  MException with properties:
```

```
    identifier: 'MATLAB:array:SizeLimitExceeded'
       message: 'Requested 281474976710655x281474976710655
 (2097152.0GB) array exceeds maximum array size preference. Creation
 of arrays greater than this limit may take a long time and cause
 MATLAB to become unresponsive. See <a href="matlab: helpview([docroot
 '/matlab/helptargets.map'], 'matlab_env_workspace_prefs')">array size
 limit</a> or preference panel for more information.'
         cause: {0x1 cell}
         stack: [4x1 struct]
    Correction: []
```

In a GraphBLAS hypersparse matrix, an m-by-n matrix A takes only O(nnz(A)) space. The difference can be huge if nnz (A) << n.

```
G = gb (huge, 1)            % no problem for GraphBLAS
H = gb (huge, huge)         % this works in GraphBLAS too
```

```
G =

    281474976710655x1 GraphBLAS double matrix, standard CSC, 0 entries


H =

    281474976710655x281474976710655 GraphBLAS double matrix,
 hypersparse CSC, 0 entries
```

Operations on huge hypersparse matrices are very fast; no component of the time or space complexity is Omega(n).

```
I = randperm (huge, 2) ;
J = randperm (huge, 2) ;
H (I,J) = 42 ;              % add 4 nonzeros to random locations in H
H = (H' * 2) ;              % transpose H and double the entries
K = gb.expand (pi, H) ;     % K = pi * spones (H)
H = H + K                   % add pi to each entry in H
numel (H)                   % this is huge^2, a really big number
```

```
H =

    281474976710655x281474976710655 GraphBLAS double matrix,
 hypersparse CSC, 4 entries

    (78390279669562,27455183225557)    87.1416
    (153933462881710,27455183225557)    87.1416
    (78390279669562,177993304104065)    87.1416
    (153933462881710,177993304104065)    87.1416


ans =
```

```
  7.9228e+28
```

All of these matrices take very little memory space:

```
whos C G H K
```

```
  Name                       Size                              Bytes  Class
  Attributes

  C          281474976710655x1                                    32  double
  sparse
  G          281474976710655x1                                   989  gb

  H          281474976710655x281474976710655                    1244  gb

  K          281474976710655x281474976710655                    1244  gb
```

The mask and accumulator

When not used in overloaded operators or built-in functions, many GraphBLAS methods of the form gb.method (...) can optionally use a mask and/or an accumulator operator. If the accumulator is '+' in gb.mxm, for example, then C = C + A*B is computed. The mask acts much like logical indexing in MATLAB. With a logical mask matrix M, C<M>=A*B allows only part of C to be assigned. If M(i,j) is true, then C(i,j) can be modified. If false, then C(i,j) is not modified.

For example, to set all values in C that are greater than 0.5 to 3, use:

```
C = rand (3)
C1 = gb.assign (C, C > 0.5, 3)      % in GraphBLAS
C (C > .5) = 3                      % in MATLAB
err = norm (C - C1, 1)
```

```
C =

    0.9575    0.9706    0.8003
    0.9649    0.9572    0.1419
    0.1576    0.4854    0.4218


C1 =

    3x3 GraphBLAS double matrix, standard CSC, 9 entries

    (1,1)    3
    (2,1)    3
    (3,1)    0.157613
    (1,2)    3
    (2,2)    3
    (3,2)    0.485376
    (1,3)    3
```

```
    (2,3)      0.141886
    (3,3)      0.421761


C =

    3.0000    3.0000    3.0000
    3.0000    3.0000    0.1419
    0.1576    0.4854    0.4218


err =

     0
```

The descriptor

Most GraphBLAS functions of the form gb.method (...) take an optional last argument, called the descriptor. It is a MATLAB struct that can modify the computations performed by the method. 'help gb.descriptorinfo' gives all the details. The following is a short summary of the primary settings:

d.out = 'default' or 'replace', clears C after the accum op is used.

d.mask = 'default' or 'complement', to use M or ~M as the mask matrix.

d.in0 = 'default' or 'transpose', to transpose A for C=A*B, C=A+B, etc.

d.in1 = 'default' or 'transpose', to transpose B for C=A*B, C=A+B, etc.

d.kind = 'default', 'gb', 'sparse', or 'full'; the output of gb.method.

```
A = sparse (rand (2)) ;
B = sparse (rand (2)) ;
C1 = A'*B ;
C2 = gb.mxm ('+.*', A, B, struct ('in0', 'transpose')) ;
err = norm (C1-C2,1)
```

```
err =

     0
```

Integer arithmetic is different in GraphBLAS

MATLAB supports integer arithmetic on its full matrices, using int8, int16, int32, int64, uint8, uint16, uint32, or uint64 data types. None of these integer data types can be used to construct a MATLAB sparse matrix, which can only be double, double complex, or logical. Furthermore, C=A*B is not defined for integer types in MATLAB, except when A and/or B are scalars.

GraphBLAS supports all of those types for its sparse matrices (except for complex, which will be added in the future). All operations are supported, including C=A*B when A or B are any integer type, for all 1,865 semirings (1,040 of which are unique).

However, integer arithmetic differs in GraphBLAS and MATLAB. In MATLAB, integer values saturate if they exceed their maximum value. In GraphBLAS, integer operators act in a modular fashion. The latter is essential when computing C=A*B over a semiring. A saturating integer operator cannot be used as a monoid since it is not associative.

The C API for GraphBLAS allows for the creation of arbitrary user-defined types, so it would be possible to create different binary operators to allow element-wise integer operations to saturate, perhaps:

```
C = gb.eadd('+saturate',A,B)
```

This would require an extension to this MATLAB interface.

```
C = uint8 (magic (3)) ;
G = gb (C) ;
C1 = C * 40
C2 = G * 40
C3 = double (G) * 40 ;
S = double (C1 < 255) ;
assert (isequal (double (C1).*S, double (C2).*S))
assert (isequal (nonzeros (C2), double (mod (nonzeros (C3), 256))))
```

```
C1 =

  3x3 uint8 matrix

   255    40   240
   120   200   255
   160   255    80


C2 =

    3x3 GraphBLAS uint8_t matrix, standard CSC, 9 entries

    (1,1)   64
    (2,1)   120
    (3,1)   160
    (1,2)   40
    (2,2)   200
    (3,2)   104
    (1,3)   240
    (2,3)   24
    (3,3)   80
```

An example graph algorithm: breadth-first search

The breadth-first search of a graph finds all nodes reachable from the source node, and their level, v. v=bfs_gb(A,s) or v=bfs_matlab(A,s) compute the same thing, but bfs_gb uses GraphBLAS matrices and operations, while bfs_matlab uses pure MATLAB operations. v is defined as v(s) = 1 for the source node, v(i) = 2 for nodes adjacent to the source, and so on.

```
clear all
rng ('default') ;
n = 1e5 ;
A = logical (sprandn (n, n, 1e-3)) ;

tic
v1 = bfs_gb (A, 1) ;
gb_time = toc ;

tic
v2 = bfs_matlab (A, 1) ;
matlab_time = toc ;

assert (isequal (full (double (v1)), v2))
fprintf ('\nnodes reached: %d of %d\n', nnz (v2), n) ;
fprintf ('GraphBLAS time: %g sec\n', gb_time) ;
fprintf ('MATLAB time:    %g sec\n', matlab_time) ;
fprintf ('Speedup of GraphBLAS over MATLAB: %g\n', ...
    matlab_time / gb_time) ;
```

```
nodes reached: 100000 of 100000
GraphBLAS time: 0.411334 sec
MATLAB time:    0.64874 sec
Speedup of GraphBLAS over MATLAB: 1.57716
```

Example graph algorithm: Luby's method in GraphBLAS

The mis_gb.m function is variant of Luby's randomized algorithm [Luby 1985]. It is a parallel method for finding an maximal independent set of nodes, where no two nodes are adjacent. See the GraphBLAS/demo/mis_gb.m function for details. The graph must be symmetric with a zero-free diagonal, so A is symmetrized first and any diagonal entries are removed.

```
A = gb (A) ;
A = A|A' ;
A = tril (A, -1) ;
A = A|A' ;

tic
s = mis_gb (A) ;
toc
fprintf ('# nodes in the graph: %g\n', size (A,1)) ;
fprintf ('# edges: : %g\n', nnz (A) / 2) ;
fprintf ('size of maximal independent set found: %g\n', ...
    full (double (sum (s)))) ;

% make sure it's independent
p = find (s == 1) ;
S = A (p,p) ;
assert (nnz (S) == 0)

% make sure it's maximal
```

```
notp = find (s == 0) ;
S = A (notp, p) ;
deg = gb.vreduce ('+.int64', S) ;
assert (logical (all (deg > 0)))
```

```
Elapsed time is 0.433782 seconds.
# nodes in the graph: 100000
# edges: : 9.9899e+06
size of maximal independent set found: 2811
```

Sparse deep neural network

The 2019 MIT GraphChallenge (see http://graphchallenge.org) is to solve a set of large sparse deep neural network problems. In this demo, the MATLAB reference solution is compared with a solution using GraphBLAS, for a randomly constructed neural network. See the dnn_gb.m and dnn_matlab.m functions for details.

```
clear all
rng ('default') ;
nlayers = 16 ;
nneurons = 4096 ;
nfeatures = 30000 ;
fprintf ('# layers:   %d\n', nlayers) ;
fprintf ('# neurons:  %d\n', nneurons) ;
fprintf ('# features: %d\n', nfeatures) ;

tic
Y0 = sprand (nfeatures, nneurons, 0.1) ;
for layer = 1:nlayers
    W {layer} = sprand (nneurons, nneurons, 0.01) * 0.2 ;
    bias {layer} = -0.2 * ones (1, nneurons) ;
end
t_setup = toc ;
fprintf ('construct problem time: %g sec\n', t_setup) ;
```

```
# layers:   16
# neurons:  4096
# features: 30000
construct problem time: 6.03621 sec
```

Solving the sparse deep neural network problem with GraphbLAS

Please wait ...

```
tic
Y1 = dnn_gb (W, bias, Y0) ;
gb_time = toc ;
fprintf ('total time in GraphBLAS: %g sec\n', gb_time) ;
```

```
setup time: 0.282851 sec
layer:    1, nnz (Y) 52031839, time 2.61494 sec
layer:    2, nnz (Y) 56297435, time 2.3141 sec
```

```
layer:    3, nnz (Y) 18532211, time 3.47658 sec
layer:    4, nnz (Y)  6388295, time 2.06636 sec
layer:    5, nnz (Y)  4773907, time 0.480592 sec
layer:    6, nnz (Y)  4429486, time 0.30974 sec
layer:    7, nnz (Y)  4350722, time 0.26216 sec
layer:    8, nnz (Y)  4329698, time 0.298956 sec
layer:    9, nnz (Y)  4320222, time 0.289372 sec
layer:   10, nnz (Y)  4318769, time 0.295618 sec
layer:   11, nnz (Y)  4317184, time 0.291833 sec
layer:   12, nnz (Y)  4317184, time 0.299612 sec
layer:   13, nnz (Y)  4317184, time 0.287857 sec
layer:   14, nnz (Y)  4317184, time 0.2886 sec
layer:   15, nnz (Y)  4317184, time 0.288256 sec
layer:   16, nnz (Y)  4317184, time 0.290996 sec
total time in GraphBLAS: 14.4509 sec
```

Solving the sparse deep neural network problem with MATLAB

Please wait ...

```
tic
Y2 = dnn_matlab (W, bias, Y0) ;
matlab_time = toc ;
fprintf ('total time in MATLAB:    %g sec\n', matlab_time) ;
fprintf ('Speedup of GraphBLAS over MATLAB: %g\n', ...
    matlab_time / gb_time) ;

err = norm (Y1-Y2,1)
```

```
layer:    1, nnz (Y) 52031843, time 15.5696 sec
layer:    2, nnz (Y) 56297445, time 16.3661 sec
layer:    3, nnz (Y) 18532218, time 20.7853 sec
layer:    4, nnz (Y)  6388296, time 10.8969 sec
layer:    5, nnz (Y)  4773911, time 2.19222 sec
layer:    6, nnz (Y)  4429487, time 1.01545 sec
layer:    7, nnz (Y)  4350725, time 0.8568 sec
layer:    8, nnz (Y)  4329700, time 1.15451 sec
layer:    9, nnz (Y)  4320224, time 1.38609 sec
layer:   10, nnz (Y)  4318775, time 1.66665 sec
layer:   11, nnz (Y)  4317184, time 3.24577 sec
layer:   12, nnz (Y)  4317184, time 1.69383 sec
layer:   13, nnz (Y)  4317184, time 1.48546 sec
layer:   14, nnz (Y)  4317184, time 1.51681 sec
layer:   15, nnz (Y)  4317184, time 1.61433 sec
layer:   16, nnz (Y)  4317184, time 1.61303 sec
total time in MATLAB:    83.784 sec
Speedup of GraphBLAS over MATLAB: 5.79784

err =

     0
```

Extreme performance differences between GraphBLAS and MATLAB.

The GraphBLAS operations used so far are perhaps 2x to 50x faster than the corresponding MATLAB operations, depending on how many cores your computer has. To run a demo illustrating a 500x or more speedup versus MATLAB, run this demo:

```
gbdemo2
```

It will illustrate an assignment C(I,J)=A that can take under a second in GraphBLAS but several minutes in MATLAB. To make the comparsion even more dramatic, try:

```
gbdemo2 (20000)
```

assuming you have enough memory. The gbdemo2 is not part of this demo since it can take a long time.

Limitations and their future solutions

The MATLAB interface for SuiteSparse:GraphBLAS is a work-in-progress. It has some limitations, most of which will be resolved over time.

(1) Nonblocking mode:

GraphBLAS has a 'non-blocking' mode, in which operations can be left pending and completed later. SuiteSparse:GraphBLAS uses the non-blocking mode to speed up a sequence of assignment operations, such as C(I,J)=A. However, in its MATLAB interface, this would require a MATLAB mexFunction to modify its inputs. That breaks the MATLAB API standard, so it cannot be safely done. As a result, using GraphBLAS via its MATLAB interface can be slower than when using its C API. This restriction would not be a limitation if GraphBLAS were to be incorporated into MATLAB itself, but there is likely no way to do this in a mexFunction interface to GraphBLAS.

(2) Complex matrices:

GraphBLAS can operate on matrices with arbitrary user-defined types and operators. The only constraint is that the type be a fixed sized typedef that can be copied with the ANSI C memcpy; variable-sized types are not yet supported. However, in this MATLAB interface, SuiteSparse:GraphBLAS has access to only predefined types, operators, and semirings. Complex types and operators will be added to this MATLAB interface in the future. They already appear in the C version of GraphBLAS, with user-defined operators in GraphBLAS/Demo/Source/usercomplex.c.

(3) Integer element-wise operations:

Integer operations in MATLAB saturate, so that uint8(255)+1 is 255. To allow for integer monoids, GraphBLAS uses modular arithmetic instead. This is the only way that C=A*B can be defined for integer semirings. However, saturating integer operators could be added in the future, so that element- wise integer operations on GraphBLAS sparse integer matrices could work just the same as their MATLAB counterparts.

So in the future, you could perhaps write this, for both sparse and dense integer matrices A and B:

```
C = gb.eadd ('+saturate.int8', A, B)
```

to compute the same thing as C=A+B in MATLAB for its full int8 matrices. % Note that MATLAB can do this only for dense integer matrices, since it doesn't support sparse integer matrices.

(4) Faster methods:

Most methods in this MATLAB interface are based on efficient parallel C functions in GraphBLAS itself, and are typically as fast or faster than the equivalent built-in operators and functions in MATLAB.

There are few notable exceptions, the most important one being horzcat and vertcat, used for [A B] and [A;B] when either A or B are GraphBLAS matrices.

Other methods that could be faster in the future include bandwidth, istriu, istril, eps, ceil, floor, round, fix, isfinite, isinf, isnan, spfun, and A.^B. These methods are currently implemented in m-functions, not in efficient parallel C functions.

```
A = sparse (rand (2000)) ;
B = sparse (rand (2000)) ;
tic
C1 = [A B] ;
matlab_time = toc ;

A = gb (A) ;
B = gb (B) ;
tic
C2 = [A B] ;
gb_time = toc ;

err = norm (C1-C2,1)
fprintf ('\nMATLAB: %g sec, GraphBLAS: %g sec\n', ...
    matlab_time, gb_time) ;
if (gb_time > matlab_time)
    fprintf ('GraphBLAS is slower by a factor of %g\n', ...
        gb_time / matlab_time) ;
end
```

```
err =

     0


MATLAB: 0.040093 sec, GraphBLAS: 0.160389 sec
GraphBLAS is slower by a factor of 4.00042
```

(5) Linear indexing:

If A is an m-by-n 2D MATLAB matrix, with n > 1, A(:) is a column vector of length m*n. The index operation A(i) accesses the ith entry in the vector A(:). This is called linear indexing in MATLAB. It is not yet available for GraphBLAS matrices in this MATLAB interface to GraphBLAS, but it could be added in the future.

(6) Implicit binary expansion

In MATLAB C=A+B where A is m-by-n and B is a 1-by-n row vector implicitly expands B to a matrix, computing C(i,j)=A(i,j)+B(j). This implicit expansion is not yet suported in GraphBLAS with C=A+B. However, it can be done with C = gb.mxm ('+.+', A, diag(gb(B))). That's an nice example of the power of semirings, but it's not immediately obvious, and not as clear a syntax as C=A+B. The GraphBLAS/demo/dnn_gb.m function uses this 'plus.plus' semiring to apply the bias to each neuron.

```
A = magic (4)
B = 1000:1000:4000
C1 = A + B
C2 = gb.mxm ('+.+', A, diag (gb (B)))
err = norm (C1-C2,1)


A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1


B =

        1000        2000        3000        4000


C1 =

        1016        2002        3003        4013
        1005        2011        3010        4008
        1009        2007        3006        4012
        1004        2014        3015        4001


C2 =

    4x4 GraphBLAS double matrix, standard CSC, 16 entries

    (1,1)    1016
    (2,1)    1005
    (3,1)    1009
    (4,1)    1004
    (1,2)    2002
    (2,2)    2011
    (3,2)    2007
    (4,2)    2014
    (1,3)    3003
    (2,3)    3010
    (3,3)    3006
    (4,3)    3015
    (1,4)    4013
    (2,4)    4008
    (3,4)    4012
    (4,4)    4001


err =

     0
```

(7) Logical indexing in subsindex and subsasgn:

The mask in GraphBLAS acts much like logical indexing in MATLAB, but it is not quite the same. Logical indexing takes the form:

```
C (M) = A (M)
```

which computes the same thing as:

```
C = gb.assign (C, M, A)
```

The gb.assign statement computes C(M)=A(M), and it is vastly faster than C(M)=A(M), even if the time to convert the gb matrix back to a MATLAB sparse matrix is included.

However, the syntax differs. The overloaded subsasgn operator for C(M)=A requires A(M) to be computed first, which becomes a 1D vector of length equal to the number of entries in M. The gb.assign function requires the original A, not the linear vector A(M). As a result, the C(M) = ... syntax is not yet supported for GraphBLAS matrices. Until I resolve this syntax issue, use C = gb.assign (C,M,A) instead.

On my 4-core Dell XPS-13 laptop, C=gb.assign(C,M,A) is several thousand times faster than C(M)=A(M) in MATLAB R2019a, so the extra syntax is well worth it. First, in GraphBLAS:

```
n = 4000 ;
tic
C = sprand (n, n, 0.1) ;
A = sparse (100 * sprand (n, n, 0.1)) ;
M = (C > 0.5) ;
t_setup = toc ;
fprintf ('\nsetup time:     %g sec\n', t_setup) ;

% even add in the time to convert C1 back to a MATLAB sparse matrix
tic
C1 = gb.assign (C, M, A) ;
C1 = double (C1) ;
gb_time = toc ;
fprintf ('\nGraphBLAS time: %g sec\n', gb_time) ;
```

```
setup time:     0.927264 sec

GraphBLAS time: 0.023827 sec
```

Please wait, this will take a few minutes or so ...

```
tic
C (M) = A (M) ;
matlab_time = toc ;

fprintf ('\nGraphBLAS time: %g sec\n', gb_time) ;
fprintf ('MATLAB time:    %g sec\n', matlab_time) ;
fprintf ('Speedup of GraphBLAS over MATLAB: %g\n', ...
    matlab_time / gb_time) ;

% GraphBLAS computes the exact same result:
assert (isequal (C1, C))
C1 - C
```

```
GraphBLAS time: 0.023827 sec
MATLAB time:    571.572 sec
Speedup of GraphBLAS over MATLAB: 23988.4

ans =

   All zero sparse: 4000x4000
```

(8) Other features are not yet in place, such as:

S = sparse (i,j,x) allows either i or j, and x, to be scalars, which are implicitly expanded. This is not yet supported by gb.build.

Many built-in functions work with GraphBLAS matrices unmodified, but sometimes things can break in odd ways. The gmres function is a built-in m-file, and works fine if given GraphBLAS matrices:

```
A = sparse (rand (4)) ;
b = sparse (rand (4,1)) ;
x = gmres (A,b)
resid = A*x-b
x = gmres (gb(A), gb(b))
resid = A*x-b
```

```
gmres converged at iteration 4 to a solution with relative residual 0.

x =

    0.0262
   -0.2499
    1.5354
   -0.4965


resid =

   1.0e-15 *

   -0.5551
   -0.2776
    0.3331
    0.0555

gmres converged at iteration 4 to a solution with relative residual 0.

x =

    0.0262
   -0.2499
    1.5354
   -0.4965


resid =
```

```
  1.0e-15 *

   0.1110
  -0.0555
   0.6661
   0.1388
```

Both of the following uses of minres (A,b) fail to converge because A is not symmetric, as the method requires. Both failures are correctly reported, and both the MATLAB version and the GraphBLAS version return the same incorrect vector x. So far so good.

```
x = minres (A, b)
[x, flag] = minres (gb(A), gb(b))
```

```
minres stopped at iteration 4 without converging to the desired
 tolerance 1e-06
because the maximum number of iterations was reached.
The iterate returned (number 4) has relative residual 0.28.

x =

    0.8201
    0.0164
    0.4958
   -0.2511


x =

    4x1 GraphBLAS double matrix, standard CSC, 4 entries

    (1,1)    0.820129
    (2,1)    0.0164381
    (3,1)    0.495776
    (4,1)    -0.251055


flag =

     1
```

But leaving off the flag output argument causes minres to try to print an error using an internal MATLAB error message utility (see 'help message'). The error message fails in an obscure way, perhaps because

```
sprintf ('%g', x)
```

fails if x is a GraphBLAS scalar. Overloading sprintf and fprintf might fix this.

```
x = minres (gb(A), gb(b))
```

```
              Array with 2 dimensions not compatible with shape of
              matrix::typed_array<double>
```

The error cannot be caught with 'try/catch' so it would terminate this demo, and thus is not attempted here. The MATLAB interface to GraphBLAS is a work-in-progress. My goal is to enable all MATLAB operations that work on MATLAB sparse matrices to also work on GraphBLAS sparse matrices, but not all methods are available yet, such as x=minres(G,b) for a GraphBLAS matrix G.

GraphBLAS operations

In addition to the overloaded operators (such as C=A*B) and overloaded functions (such as L=tril(A)), GraphBLAS also has methods of the form gb.method, listed on the next page. Most of them take an optional input matrix Cin, which is the initial value of the matrix C for the expression below, an optional mask matrix M, and an optional accumulator operator.

```
C<#M,replace> = accum (C, T)
```

In the above expression, #M is either empty (no mask), M (with a mask matrix) or ~M (with a complemented mask matrix), as determined by the descriptor. 'replace' can be used to clear C after it is used in accum(C,T) but before it is assigned with C<...> = Z, where Z=accum(C,T). The matrix T is the result of some operation, such as T=A*B for gb.mxm, or T=op(A,B) for gb.eadd.

A short summary of these gb.methods is on the next page.

List of gb.methods

```
gb.clear                     clear GraphBLAS workspace and settings
gb.descriptorinfo (d)        list properties of a descriptor d
gb.unopinfo (op, type)       list properties of a unary operator
gb.binopinfo (op, type)      list properties of a binary operator
gb.monoidinfo (op, type)     list properties of a monoid
gb.semiringinfo (s, type)    list properties of a semiring
t = gb.threads (t)           set/get # of threads to use in GraphBLAS
c = gb.chunk (c)             set/get chunk size to use in GraphBLAS
e = gb.nvals (A)             number of entries in a matrix
G = gb.empty (m, n)          return an empty GraphBLAS matrix
s = gb.type (X)              get the type of a MATLAB or gb matrix X
f = gb.format (f)            set/get matrix format to use in GraphBLAS
C = expand (scalar, S)       expand a scalar (C = scalar*spones(S))

G = gb.build (I, J, X, m, n, dup, type, d)      build a matrix
[I,J,X] = gb.extracttuples (A, d)               extract all entries

C = gb.mxm (Cin, M, accum, semiring, A, B, d)   matrix multiply
C = gb.select (Cin, M, accum, op, A, thunk, d)  select entries
C = gb.assign (Cin, M, accum, A, I, J, d)       assign, like C(I,J)=A
C = gb.subassign (Cin, M, accum, A, I, J, d)    assign, different M
C = gb.vreduce (Cin, M, accum, op, A, d)        reduce to vector
C = gb.reduce (Cin, accum, op, A, d)            reduce to scalar
C = gb.gbkron (Cin, M, accum, op, A, B, d)      Kronecker product
C = gb.gbtranspose (Cin, M, accum, A, d)        transpose
C = gb.eadd (Cin, M, accum, op, A, B, d)        element-wise addition
C = gb.emult (Cin, M, accum, op, A, B, d)       element-wise mult.
C = gb.apply (Cin, M, accum, op, A, d)          apply unary operator
C = gb.extract (Cin, M, accum, A, I, J, d)      extract, like C=A(I,J)
```

For more details type 'help graphblas' or 'help gb'.

Tim Davis, Texas A&M University, http://faculty.cse.tamu.edu/davis See also sparse, doc sparse, and https://twitter.com/DocSparse

Published with MATLAB® R2019a