A Hands-On Introduction to the GraphBLAS

http://graphblas.org

Scott McMillan
CMU/SEI

Tim Mattson
Intel Labs

Brought to you by the “GraphBLAS C Specification Gang” (of Five):
Aydın Buluç (LBNL), Tim Mattson (Intel), Scott McMillan (CMU/SEI)
Jose Moreira (IBM), Carl Yang (UC Davis)

… and a special thank you to Tim Davis (Texas A&M) for GraphBLAS support in SuiteSparse
HPEC tutorial

• We had 16 students by the end of the day.
• Two of them had Windows systems and were not able to fully participate. They stayed through the whole tutorial (I guess they did the exercises but didn’t compiler or run them).
• One person (not included in that count of 16) at the beginning of the tutorial when he heard we do not support windows, left to attend a different tutorial.
• We took about two hours to get up to the section on Breadth first traversal which is pretty much exactly what we expected. We did skip exercise 6 which was just fine.
• We made it to the end of the tutorial. Some students actually finished the BFS levels exercise.
Outline

- Graphs and Linear Algebra
- The GraphBLAS C API and Adjacency Matrices
- GraphBLAS Operations
- Breadth-First Traversal
Understanding relationships between items

• Graph: A visual representation of a set of vertices and the connections between them (edges).

• Graph: Two sets, one for the vertices \( (v) \) and one for the edges \( (e) \)

\[
v \in [0, 1, 2, 3, 4, 5, 6]
\]

\[
e \in [(0,1), (0,3), (1,4), (1,6), (2,5), (3,0), (3,2), (4,5), (5,2), (6,2), (6,3), (6,4)]
\]
A graph as a matrix

• Adjacency Matrix: A square matrix (usually sparse) where rows and columns are labeled by vertices and non-empty values are edges from a row vertex to a column vertex.

By using a matrix, I can turn algorithms working with graphs into linear algebra.
Graph Algorithms and Linear Algebra

- Most common graph algorithms can be represented in terms of linear algebra.
  - This is a mature field … it even has a book.

- Benefits of graphs as linear algebra
  - Well suited to memory hierarchies of modern microprocessors
  - Can utilize decades of experience in distributed/parallel computing from linear algebra in supercomputing.
  - Easier to understand … for some people.
How do linear algebra people write software?

- They do so in terms of the BLAS:
  - The Basic Linear Algebra Subprograms: low-level building blocks from which any linear algebra algorithm can be written

<table>
<thead>
<tr>
<th>BLAS 1</th>
<th>Vector/vector</th>
<th>Lawson, Hanson, Kincaid and Krogh, 1979</th>
<th>LINPACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS 2</td>
<td>Matrix/vector</td>
<td>Dongarra, Du Croz, Hammarling and Hanson, 1988</td>
<td>LINPACK on vector machines</td>
</tr>
<tr>
<td>BLAS 3</td>
<td>Matrix/matrix</td>
<td>Dongarra, Du Croz, Hammarling and Hanson, 1990</td>
<td>LAPACK on cache based machines</td>
</tr>
</tbody>
</table>

- The BLAS supports a separation of concerns:
  - HW/SW optimization experts tuned the BLAS for specific platforms.
  - Linear algebra experts built software on top of the BLAS ... high performance “for free”.

- It is difficult to over-state the impact of the BLAS ... they revolutionized the practice of computational linear algebra.
GraphBLAS: building blocks for graphs as linear algebra

- Basic objects
  - Matrix, vector, algebraic structures, and "control objects"

- Fundamental operations over these objects

Matrix multiplication

Matrix-vector multiplication (vxm, mxv)

Element-wise operations (eWiseAdd, eWiseMult)

Extract (and Assign) submatrices

...plus reductions, transpose, and application of a function to each element of a matrix or vector
GraphBLAS References

Mathematical Foundations of the GraphBLAS

Jeremy Kepner (MIT Lincoln Laboratory Supercomputing Center), Peter Aaltonen (Indiana University), David Bader (Georgia Institute of Technology), Aydın Buluç (Lawrence Berkeley National Laboratory), Franz Franchetti (Carnegie Mellon University), John Gilbert (University of California, Santa Barbara), Dylan Hutchison (University of Washington), Manoj Kumar (IBM), Andrew Lumsdaine (Indiana University), Henning Meyerhenke (Karlsruhe Institute of Technology), Scott McMillan (CMU Software Engineering Institute), Jose Moreira (IBM), John D. Owens (University of California, Davis), Carl Yang (University of California, Davis), Marcin Zalewski (Indiana University), Timothy Mattson (Intel)

Design of the GraphBLAS API for C

Aydın Buluç†, Tim Mattson‡, Scott McMillan§, José Moreira¶, Carl Yang*,†

†Computational Research Division, Lawrence Berkeley National Laboratory
‡Intel Corporation
§Software Engineering Institute, Carnegie Mellon University
¶IBM Corporation
*Electrical and Computer Engineering Department, University of California, Davis, USA

The official GraphBLAS C spec can be found at: www.graphblas.org
GraphBLAS Implementations

• Multiple implementation projects:
  – A C-wraper around the GPU Gunrock library from UC Davis, http://adsabs.harvard.edu/abs/2017arXiv170101170W
  – The IBM GraphBLAS C implementation, https://github.com/IBM/ibmgraphblas
  – CMU/SEI C++ GraphBLAS Template Library https://github.com/cmu-sei/gbtl (there is also a python wrapper called pyGB with UW/PNNL).
  
• We’ll use SuiteSparse:GraphBLAS in this tutorial
Exercise 1: Build a GraphBLAS program

- Clone our git repository
- Includes the following components
  - Exercises and solutions
  - SuiteSparse library, binaries for Linux and OSX and source
- Load software onto your system, make sure you can build and run our test program

$ git clone https://github.com/tgmattso/GraphBLAS.git
$ cd GraphBLAS/src
$ make BuildGraph.exe
$ ./BuildGraph.exe

- If all goes well, your output should look like this:

  $ ./BuildGraph.exe
  Matrix: GRAPH =
  [ - , - , - ]
  [ - , - , 4 ]
  [ - , - , - ]
Outline

• Graphs and Linear Algebra
• The GraphBLAS C API and Adjacency Matrices
• GraphBLAS Operations
• Breadth-First Traversal
GraphBLAS C API

- A binding of the GraphBLAS math to the C programming language.

- Requires C99 extended with function polymorphism based on static-types and number-of-parameters.
  - All modern C compilers in common use today support these extensions

- Basic include file with function prototypes, types, and constants
  - `#include <GraphBLAS.h>`

- Includes a few types and opaque objects (e.g. matrices and vectors) to give implementations maximum flexibility

  - `GrB_Index`  →  An integer type used to set dimensions and index into arrays
  - `GrB_Matrix` →  A 2D sparse array, row indices, column indices and values
  - `GrB_Vector` →  A 1D sparse array

  - … plus additional opaque objects we’ll describe later (descriptors, semirings, binary operators, and unary operators)
GraphBLAS C API: Basic definitions

- **Opaque object**: An object manipulated strictly through the GraphBLAS API whose implementation is not defined by the GraphBLAS specification.

- **Transparent object**: an object whose structure is fully exposed to the programmer. E.g.: an array of tuples <i, j, value>

- **Method**: Any C function that manipulates a GraphBLAS opaque object.

- **Domain**: the set of available values used for the elements of matrices, the elements of vectors, and when defining operators.
  - Examples are `GrB_UINT64`, `GrB_INT32`, `GrB_BOOL`, `GrB_FP32`

- **Operation**: a method that corresponds to an operation defined in the GraphBLAS math spec. [http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf](http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf)
  - Examples: matrix multiply, matrix-vector multiply, reduction, apply
Execution modes

• A GraphBLAS program defines a DAG of operations.
• Objects are defined by the sequence of GraphBLAS method calls, but the value of the object is not assured until a GraphBLAS method queries its state.
• This gives an implementation flexibility to optimize the execution (fusing methods, replacing method sequences by more efficient ones, etc.)

GrB_op1(A);
GrB_op2(B);
GrB_op3(C,A,B);

• An execution of a GraphBLAS program defines a context for the library.
• The execution runs in one of two modes:
  – Blocking mode … executes methods in program order with each method completing before the next is called
  – Non-Blocking mode … methods launched in order. Complete in any order consistent with the DAG. Objects do not exit in fully defined state until queried.

• Most implementations only support blocking mode.
Predefined low-level types

- Predefined types used to define domains in GraphBLAS

<table>
<thead>
<tr>
<th>GrB_Type values</th>
<th>C type</th>
<th>domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_BOOL</td>
<td>bool</td>
<td>{false, true}</td>
</tr>
<tr>
<td>GrB_INT8</td>
<td>int8_t</td>
<td>\mathbb{Z} \cap [-2^7, 2^7)</td>
</tr>
<tr>
<td>GrB_UINT8</td>
<td>uint8_t</td>
<td>\mathbb{Z} \cap [0, 2^8)</td>
</tr>
<tr>
<td>GrB_INT16</td>
<td>int16_t</td>
<td>\mathbb{Z} \cap [-2^{15}, 2^{15})</td>
</tr>
<tr>
<td>GrB_UINT16</td>
<td>uint16_t</td>
<td>\mathbb{Z} \cap [0, 2^{16})</td>
</tr>
<tr>
<td>GrB_INT32</td>
<td>int32_t</td>
<td>\mathbb{Z} \cap [-2^{31}, 2^{31})</td>
</tr>
<tr>
<td>GrB_UINT32</td>
<td>uint32_t</td>
<td>\mathbb{Z} \cap [0, 2^{32})</td>
</tr>
<tr>
<td>GrB_INT64</td>
<td>int64_t</td>
<td>\mathbb{Z} \cap [-2^{63}, 2^{63})</td>
</tr>
<tr>
<td>GrB_UINT64</td>
<td>uint64_t</td>
<td>\mathbb{Z} \cap [0, 2^{64})</td>
</tr>
<tr>
<td>GrB_FP32</td>
<td>float</td>
<td>IEEE 754 binary32</td>
</tr>
<tr>
<td>GrB_FP64</td>
<td>double</td>
<td>IEEE 754 binary64</td>
</tr>
</tbody>
</table>
int main(int argc, char** argv) {
    GrB_init(GrB_BLOCKING);
    GrB_Index const NUM_NODES = 3;
    GrB_Matrix graph;
    GrB_Matrix_new(&graph, GrB_UINT64,
                   NUM_NODES, NUM_NODES);
    GrB_Matrix_setElement(graph, 4, 1, 2);

    pretty_print_matrix_UINT64(graph, "GRAPH");
    GrB_Index nvals;
    GrB_Matrix_nvals(&nvals, graph);
    assert(nvals == 1);

    // Cleanup
    GrB_free(&graph);
    GrB_finalize();
}
Exercise 2: Adjacency matrix

• Draw a simple graph with 3 to 5 nodes.
• Write a program to create the adjacency matrix.
  – Use BuildGraph.c as an example.
• Output the result and verify that your adjacency graph is correct.
• You will need the following types and methods from the GraphBLAS
  – GrB_Index, GrB_Matrix
  – GrB_init(); GrB_finalize();
  – GrB_Matrix_new(&graph, GrB_domain, Nrows, Ncols);
  – GrB_Matrix_setElement(graph, value, from_node, to_node);
  – GrB_Matrix_nvals(&nvals, graph);
  – GrB_free(&graph);

• Hint: Save time and minimize typing
  – Copy BuildGraph.c into another file (e.g. exercise2.c) and modify it to build
    your adjacency matrix program.
  – Edit the makefile and add your new source file to the list in the definition of
    SOURCES. Then you can just type “make” to build your program.
Exercise 2: Adjacency matrix

- Draw a simple graph with 3 to 5 nodes.
- Write a program to create the adjacency matrix.
  - Use BuildGraph.c as an example.
- Output the result and verify that your adjacency graph is correct.
- You will need the following types and methods from the GraphBLAS
  - `GrB_Index`, `GrB_Matrix`
  - `GrB_init();` `GrB_finalize();`
  - `GrB_Matrix_new(&graph, GrB_domain, Nrows, Ncols);`
  - `GrB_Matrix_setElement(graph, value, from_node, to_node);`
  - `GrB_Matrix_nvals(&nvals, graph);`
  - `GrB_free(&graph);`

- Hint: Save time and minimize typing
  - Copy BuildGraph.c into another file (e.g. exercise1.c) and modify it to build your adjacency matrix program.
  - Edit the makefile and add your new source file to the list in the definition of SOURCES. Then you can just type “make” to build your program.
Solution to Exercise 2

...  
GrB_init(GrB_BLOCKING);

GrB_Index const NUM_NODES = 3;
GrB_Matrix graph;
GrB_Matrix_new(&graph, GrB_UINT64,
               NUM_NODES, NUM_NODES);

GrB_Matrix_setElement(graph, 4, 1, 2);
GrB_Matrix_setElement(graph, 4, 2, 1);
GrB_Matrix_setElement(graph, 2, 0, 1);
GrB_Matrix_setElement(graph, 2, 1, 0);

pretty_print_matrix_UINT64(graph, "Graph");

GrB_free(&graph);
GrB_finalize();

Our three node graph with edge weights:

Matrix: Graph:
[ - , 2 , - ]
[ 2 , - , 4 ]
[ - , 4 , - ]
Building matrices

• Building a matrix one edge at a time is awkward.
• It is often more convenient to do it from vectors defining the indices and values for non-empty elements of the sparse matrix

```c
GrB_Info GrB_Matrix_build( GrB_Matrix C,
                           const GrB_Index *row_indices,
                           const GrB_Index *col_indices,
                           const <type>*values,
                           GrB_Index n,
                           const GrB_BinaryOp dup);
```

• `row_indices`, `col_indices`, and `values` are transparent arrays.
• `<type>` is a C type consistent with the domain of the matrix
• `n` is the number of entries in the sparse matrix
• `dup` is an associative, commutative function to apply to the values should duplicate locations be specified.
  - Typically use one of the GraphBLAS predefined operators
Building matrices

- Building a matrix one edge at a time is awkward.
- It is often more convenient to do it from vectors defining the indices and values for non-empty elements of the sparse matrix.

```c
GrB_Info GrB_Matrix_build( GrB_Matrix C,
                          const GrB_Index *row_indices,
                          const GrB_Index *col_indices,
                          const <type> *values,
                          GrB_Index n,
                          const GrB_BinaryOp dup);
```

- `row_indices`, `col_indices`, and `values` are transparent arrays.
- `<type>` is a C type consistent with the domain of the matrix.
- `n` is the number of entries in the sparse matrix.
- `dup` is an associative, commutative function to apply to the values should duplicate locations be specified.
  - Typically use one of the GraphBLAS predefined operators.
GraphBLAS predefined operators

- A subset of operators from Table 2.3 of the GraphBLAS specification

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Domains</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_LOR</td>
<td>bool x bool → bool</td>
<td>f(x,y) = x ∨ y</td>
</tr>
<tr>
<td>GrB_LAND</td>
<td>bool x bool → bool</td>
<td>f(x,y) = x ∧ y</td>
</tr>
<tr>
<td>GrB_EQ_T</td>
<td>T x T → bool</td>
<td>f(x,y) = (x==y)</td>
</tr>
<tr>
<td>GrB_MIN_T</td>
<td>T x T → T</td>
<td>f(x,y) = (x&lt;y)?x:y</td>
</tr>
<tr>
<td>GrB_MAX_T</td>
<td>T x T → T</td>
<td>f(x,y) = (x&gt;y)?x:y</td>
</tr>
<tr>
<td>GrB_PLUS_T</td>
<td>T x T → T</td>
<td>f(x,y) = x + y</td>
</tr>
<tr>
<td>GrB_TIMES_T</td>
<td>T x T → T</td>
<td>f(x,y) = x * y</td>
</tr>
<tr>
<td>GrB_FIRST_T</td>
<td>T x T → T</td>
<td>f(x,y) = x</td>
</tr>
<tr>
<td>GrB_SECOND_T</td>
<td>T x T → T</td>
<td>f(x,y) = y</td>
</tr>
</tbody>
</table>

Where \( T \) is a suffix indicating type and includes \( \text{FP32} \), \( \text{FP64} \), \( \text{INT32} \), \( \text{UINT32} \), \( \text{BOOL} \).

Note: \( \text{GrB\_FIRST} \) and \( \text{GrB\_SECOND} \) are not commutative operators.

This is a subset of the defined types and operators. See table 2.3 for the full list.
C code fragment using GrB_Matrix_build

GrB_Index const NUM_NODES = 3;
GrB_Index const NUM_EDGES = 4;
GrB_Index row_indices[] = {0, 1, 1, 2};
GrB_Index col_indices[] = {1, 0, 2, 1};
bool values[] = {true, true, true, true, true};
GrB_Matrix graph;

GrB_Matrix_new(&graph, GrB_BOOL, NUM_NODES, NUM_NODES);

GrB_Matrix_build(graph,
    row_indices, col_indices, (bool*)values,
    NUM_EDGES, GrB_LOR);
Exercise 3: Adjacency matrix

• Write a program to create the adjacency matrix for the GraphBLAS “logo” graph using row, column and value arrays.

• You will need the following types and methods from the GraphBLAS
  - GrB_Index, GrB_Matrix
  - GrB_init(); GrB_finalize();
  - GrB_Matrix_new(&graph, GrB_domain, Nrows, Ncols);
  - GrB_Matrix_build(graph, row_indices, col_indices, values, NUM_EDGES, dup);
  - GrB_Matrix_nvals(&nvals, graph);
  - GrB_free(&graph);
Summary of solution to exercise 3

GrB_Index const NUM_NODES = 7;
GrB_Index const NUM_EDGES = 12;

GrB_Index row_indices[] = {0, 0, 1, 1, 2, 3, 3, 4, 5, 6, 6, 6};
GrB_Index col_indices[] = {1, 3, 4, 6, 5, 0, 2, 5, 2, 2, 3, 4};
bool values[] = {true, true, true, true, true, true, true,
                 true, true, true, true, true, true};

GrB_Matrix graph;
GrB_Matrix_new(&graph, GrB_BOOL, NUM_NODES, NUM_NODES);
GrB_Matrix_build(graph, row_indices, col_indices, (bool*)values,
                 NUM_EDGES, GrB_LOR);
pretty_print_matrix_UINT64(graph, "Graph");
Outline

• Graphs and Linear Algebra
• The GraphBLAS C API and Adjacency Matrices
• GraphBLAS Operations
• Breadth-First Traversal
# GraphBLAS Operations (from the Math Spec*)

<table>
<thead>
<tr>
<th>Operation name</th>
<th>Mathematical description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxm</td>
<td>$C \odot A \oplus B$</td>
</tr>
<tr>
<td>mxv</td>
<td>$w \odot A \oplus v$</td>
</tr>
<tr>
<td>vxm</td>
<td>$w^T \odot v^T \oplus A$</td>
</tr>
<tr>
<td>eWiseMult</td>
<td>$C \odot A \otimes B$</td>
</tr>
<tr>
<td>eWiseAdd</td>
<td>$w \odot u \otimes v$</td>
</tr>
<tr>
<td>reduce (row)</td>
<td>$w \odot \bigoplus_j A(:,j)$</td>
</tr>
<tr>
<td>apply</td>
<td>$C \odot F_u(A)$</td>
</tr>
<tr>
<td>transpose</td>
<td>$C \odot A^T$</td>
</tr>
<tr>
<td>extract</td>
<td>$C \odot A(i,j)$</td>
</tr>
<tr>
<td>assign</td>
<td>$C(i,j) \odot A$</td>
</tr>
</tbody>
</table>

We use $\odot$, $\oplus$, and $\otimes$ since later on we’ll manipulate the algebraic structure to generalize them to other operations.

* Mathematical foundations of the GraphBLAS, Kepner et. al. HPEC’2016
Multiply a matrix times a vector to produce a vector

\[ w(i) = w(i) \odot \sum_{k=0}^{N} A(i, k) \otimes u(k) \]

Definitions:
- \( S \) is the domain of the objects \( w, u, \) and \( A \)
- \( \odot \) is an optional accumulation operator (a binary operator)
- \( \otimes \) and \( \oplus \) are multiplication and addition (or generalizations thereof)
- \( \Sigma \) uses the \( \oplus \) operator
Multiply a matrix times a vector to produce a vector

\[ w(i) = w(i) \odot \sum_{k \in \text{ind}(A(i,:)) \cap \text{ind}(u)} A(i, k) \otimes u(k) \]

The summation is over the intersection of the existing elements in the \(i^{th}\) row of \(A\) with \(u\) ... which avoids exposing how empty elements (i.e. “zeros”) are represented. This becomes important when we change the semiring between operations

\[ w \in S^M \quad u \in S^N \quad A \in S^{M \times N} \]

Definitions:
- \(S\) is the domain of the objects \(w, u,\) and \(A\)
- \(\odot\) is an optional accumulation operator (a binary operator)
- \(\otimes\) and \(\oplus\) are multiplication and addition (or generalizations thereof)
- \(\Sigma\) uses the \(\oplus\) operator
- \(\text{ind}(u)\) returns the indices of the stored values of \(u\)
GrB_mxv()

- Compute the product of a GraphBLAS sparse matrix with a GraphBLAS vector.
- Returns error codes of type GrB_info. See the spec for details.

GrB_Info GrB_mxv(GrB_Vector w, const GrB_Vector mask, const GrB_BinaryOp accum, const GrB_Semiring op, const GrB_Matrix A, const GrB_Vector u, const GrB_Descriptor desc);
GrB_mxv()  \[ w \odot = A \odot \otimes u \]

- Compute the product of a GraphBLAS sparse matrix with a GraphBLAS vector.
- Returns error codes of type GrB_info. See the spec for details.

\[
\text{GrB_INFO \ GrB_mxv}(\text{GrB_Vector } w, \text{const GrB_Vector mask, \text{const GrB_BinaryOp accum, \text{const GrB_Semiring op, \text{const GrB_Mat}rix A, \text{const GrB_Vector u, \text{const GrB_Descriptor desc)}}}) \rightarrow \text{GrB.NULL}
\]

Let’s ignore mask, accum and desc for now and use default values (indicated by GrB_NULL)
\textbf{GrB\_mxv()}\hfill \textbf{w \blacktimes = A \bigoplus \cdot \blacktriangleright u}

- Compute the product of a GraphBLAS sparse matrix with a GraphBLAS vector.
- Returns error codes of type \texttt{GrB\_info}. See the spec for details.

\begin{verbatim}
GrB\_Info GrB\_mxv(GrB\_Vector w, const GrB\_Vector mask, const GrB\_BinaryOp accum, const GrB\_Semiring op, const GrB\_Matrix A, const GrB\_Vector u, const GrB\_Descriptor desc) {
    return GrB\_NULL;
}
\end{verbatim}

\texttt{Op} defines the algebraic structure, a semiring in this case. This gives us \( \blacktimes \) and \( \bigoplus \) and the identity for \( \bigoplus \). We’ll say much more about this later.

For our first exercises with bool objects, we’ll use a built-in SuiteSparse semiring \texttt{GxB\_LOR\_LAND\_BOOL}.
Exercise 4: Matrix Vector Multiplication

• Use the adjacency matrix from exercise 3 and a vector with a single value to select one of the nodes in the graph.
• Find the product $mxv$, print the result, and interpret its meaning.
• In addition to those from Exercise 3, you’ll need the functions:
  
  - `GrB_Vector result, vec;`
  - `GrB_Index NODE;`
  - `GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);`
  - `GrB_Vector_setElement(vec, true, NODE);`
  - `pretty_print_vector_UINT64(vec, "Input node");`
  - `GrB_mxv(result, GrB_NULL, GrB_NULL, GxB_LOR_LAND_BOOL, graph, vec, GrB_NULL);`
Solution to exercise 4

```c
pretty_print_matrix_UINT64(graph, "GRAPH");

// Build a vector with one node set.
GrB_Index const NODE = 2;
GrB_Vector vec, result;
GrB_Vector_new(&result, GrB_BOOL, NUM_NODES);
GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(vec, true, NODE);
pretty_print_vector_UINT64(vec, "Target node");

GrB_mxv(result, GrB_NULL, GrB_NULL,
         GxB_LOR_LAND_BOOL, graph, vec, GrB_NULL);
pretty_print_vector_UINT64(result, "sources");
```

The stored elements of the adjacency matrix, \( a(i,j) \) indicate an edge from vertex \( i \) to vertex \( j \).

So the matrix vector product scans over a row (from) to find when an edge lands at the destination.
Finding neighbors

• A more common operation is to input a vector selecting a source and find all the neighbors one hop away from that vertex.
• Using `mxv()`, how would you do this?
Finding neighbors

• A more common operation is to input a vector selecting a source and find all the neighbors one hop away from that vertex.
• Using mxv(), how would you do this?
  – The adjacency matrix elements indicate edges
    – from a vertex (row index)
    – to another vertex (columns index)
  – Then the transpose of the adjacency matrix indicates edges
    – To a vertex (row index)
    – From other vertices (column index)
• Therefore, we can find the neighbors of a vertex (marked by the non-empty elements of v)

\[ \text{Neighbors} = A^T \oplus \otimes v \]

• The GraphBLAS defines a transpose operation, but given how often you need to do a transpose, there must be a better way
Changing the behavior of a GraphBLAS operation

• Most GraphBLAS operations take an argument that is an opaque object called a “descriptor”. You declare an descriptor called ”desc” and create it as follows:
  
  ```
  GrB_Descriptor desc;
  GrB_Descriptor_new (&desc);
  ```

• The descriptor controls the behavior of the method and how objects are handled inside the method.

• The descriptor controls:
  
  – Do you transpose input matrices? (GrB_TRAN)
  – Does the computation replace existing values in the output object or combine with them? (GrB_REPLACE)
  – Take the structural complement of the mask object (swap empty/false $$\leftrightarrow$$ filled/true values in a sparse object). (GrB_SCMP)

  ....To be discussed later
Using Descriptors

• A descriptor is an opaque object so you set its values with a GraphBLAS method.

• A descriptor field selects the object it impacts:
  – \texttt{GrB\_OUTP}: The output GraphBLAS object
  – \texttt{GrB\_INP0}: The first input GraphBLAS object (matrix or vector)
  – \texttt{GrB\_INP1}: The second input GraphBLAS object (matrix or vector)
  – \texttt{GrB\_MASK}: The GraphBLAS mask object (described later).

• A descriptor value describes the action to be taken.

• For example, to transpose the first input matrix, you’d call the operation and pass in the following descriptor
  \begin{verbatim}
  GrB_Descriptor desc;
  GrB_Descriptor_new(&desc);
  GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);
  \end{verbatim}
Exercise 5: Matrix Vector Multiplication

- Modify your program from exercise 4 to multiply by the transpose of the adjacency matrix.
- Verify that you can use that to find the one-hop neighbors of any vertex

- GrB_Vector result, vec;
- GrB_Index NODE;
- GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);
- GrB_Vector_setElement(vec, true, NODE);
- pretty_print_vector_UINT64(vec, "Input node");
- GrB_Descriptor desc;
- GrB_Descriptor_new(&desc);
- GrB_Descriptor_set(desc, FIELD, VALUE)
- GrB_mxv(result, GrB_NULL, GrB_NULL,
          GxB_LOR_LAND_BOOL, graph, vec, desc);

FIELD: GrB_INP0, GrB_INP1, GrB_OUTP, GrB_MASK
VALUE: GrB_TRAN, GrB_REPLACE, GrB_SCMP
// Build a vector with one node set.
GrB_Index const SRC_NODE = 6;
GrB_Vector vec;
GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(vec, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);

pretty_print_vector_UINT64(vec, "source node");
GrB_mxv(vec, GrB_NULL, GrB_NULL,
        GxB_LOR_LAND_BOOL, graph, vec, desc);
pretty_print_vector_UINT64(vec, "neighbors");

Vector: source node =
[ - - - - - - 1 ]  
Vector: neighbors =
[ - - 1 1 1 - - ]
GrB_mxv test passed.

The transposed matrix vector product scans over a columns (to) to find edges that start at the source node.
GrB_mxv() computes the product of a GraphBLAS Sparse Matrix with a GraphBLAS vector. Returns error codes of type GrB_info. See the spec for details.

\[ w \odot = A \oplus \odot u \]

It's time to explain semirings in GraphBLAS operation.
Algebraic Semirings

• Semiring: An Algebraic structure that generalizes real arithmetic by replacing $(+,*)$ with binary operations $(\text{Op1}, \text{Op2})$
  – Op1 and Op2 have identity elements sometimes called 0 and 1
  – Op1 and Op2 are associative.
  – Op1 is commutative, Op2 distributes over Op1 from both left and right
  – The Op1 identify is an Op2 annihilator.
Algebraic Semirings

- Semiring: An Algebraic structure that generalizes real arithmetic by replacing (+,*) with binary operations (Op1, Op2)
  - Op1 and Op2 have identity elements sometimes called 0 and 1
  - Op1 and Op2 are associative.
  - Op1 is commutative, Op2 distributes over Op1 from both left and right
  - The Op1 identity is an Op2 annihilator.

<table>
<thead>
<tr>
<th>(R, +, *, 0, 1)</th>
<th>Standard operations in linear algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Field</td>
<td></td>
</tr>
</tbody>
</table>

Notation: \((R, +, *, 0, 1)\)

<table>
<thead>
<tr>
<th>Scalar type</th>
<th>Op1</th>
<th>Op2</th>
<th>Identity Op1</th>
<th>Identity Op2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algebraic Semirings

- Semiring: An Algebraic structure that generalizes real arithmetic by replacing (+,*) with binary operations (Op1, Op2)
  - Op1 and Op2 have identity elements sometimes called 0 and 1
  - Op1 and Op2 are associative.
  - Op1 is commutative, Op2 distributes over Op1 from both left and right
  - The Op1 identify is an Op2 annihilator.

<table>
<thead>
<tr>
<th>(R, +, *, 0, 1) Real Field</th>
<th>Standard operations in linear algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R U {∞}, min, +, ∞, 0) Tropical semiring</td>
<td>Shortest path algorithms</td>
</tr>
<tr>
<td>({0,1},</td>
<td>, &amp;, 0, 1) Boolean Semiring</td>
</tr>
<tr>
<td>(R U {∞}, min, *, ∞, 1)</td>
<td>Selecting a subgraph or contracting nodes to form a quotient graph.</td>
</tr>
</tbody>
</table>
Algebraic structures in the GraphBLAS: Semirings and Monoids

• The GraphBLAS semiring defines:
  – A set of allowed values (the domain)
  – Two commutative operators called addition and multiplication
  – An additive identity (called 0) that is the annihilator over multiplication.

• A Monoid is used in defining a semiring:
  – Monoid: A domain, an associative binary operator and an identity corresponding to that operator

Hierarchy of algebraic object classes showing relationships between the various domains and the operators.
Building Semirings in the GraphBLAS

• First you build the monoid (M) for a particular domain, D, the “addition” operator, and its identity:

\[ M = < D, \oplus, 0 > \]

• Then define the semiring (S) in terms of the Monoid and the multiplications operator:

\[ S = < D_{out}, D_{in1}, D_{in2}, M, \otimes > \]

• The domains must be consistent:

\[ \otimes: D_{in1} \times D_{in2} \rightarrow D_{out} \]
\[ \oplus: D_{out} \times D_{out} \rightarrow D_{out} \]
\[ 0 \in D_{out} \]
Building Semirings in the GraphBLAS

• First you build the monoid (M) for the “addition” and its identity:

\[
\text{GrB\_Info GrB\_Monoid\_new}(\text{GrB\_Monoid } \ast\text{monoid, GrB\_BinaryOp binary\_op, <type> identity});
\]

• Where the type must be consistent with that of the binary operator which is either a built-in operator (Spec. Table 2.3) or a user-defined operator (not covered here)

• Example:

\[
\text{GrB\_Monoid UInt64Plus } \; \\
\text{GrB\_Monoid\_new(&UInt64Plus, GrB\_PLUS\_UINT64, 0 ul)};
\]
Building Semirings in the GraphBLAS

• Then you build the semiring pairing a monoid (“add”) with a binary operator (“mul”):

\[
\text{GrB\_Info GrB\_Semiring\_new}(\text{GrB\_Semiring } \text{*semiring}, \\
\text{GrB\_Monoid } \text{add\_op}, \\
\text{GrB\_BinaryOp } \text{mul\_op});
\]

• The monoid’s identity \textit{should} be the binary operator’s annihilator (not enforced).

• Example using the monoid from the previous page:

```c
GrB\_Semiring UInt64Arith;
GrB\_Semiring\_new(&UInt64Arith, UInt64Plus, GrB\_TIMES\_UINT64);
```
# Common Semirings

<table>
<thead>
<tr>
<th>semiring</th>
<th>Domain</th>
<th>Add</th>
<th>Add-identity</th>
<th>multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>GrB_BOOL</td>
<td>GrB_LOR</td>
<td>false</td>
<td>GrB_LAND</td>
</tr>
<tr>
<td>Int32 arithmetic</td>
<td>GrB_INT32</td>
<td>GrB_PLUS_INT32</td>
<td>0</td>
<td>GrB_TIMES_INT32</td>
</tr>
<tr>
<td>FP32 arithmetic</td>
<td>GrB_FP32</td>
<td>GrB_PLUS_FP32</td>
<td>0.0f</td>
<td>GrB_TIMES_FP32</td>
</tr>
<tr>
<td>Max_second</td>
<td>GrB_FP32</td>
<td>GrB_MAX_FP32</td>
<td>0.0f</td>
<td>GrB_SECOND_FP32</td>
</tr>
</tbody>
</table>
Exercise 6: Changing semirings

• Up to this point, we’ve used a built-in Boolean semiring that is included with SuiteSparse (GxB_LOR_LAND_BOOL).
• Pick any of the past exercises and experiment with different semi-rings.
  - GrB_Monoid UInt64Plus;
  - GrB_Monoid_new(&UInt64Plus, GrB_PLUS_UINT64, 0ul);
  - GrB_Semiring UInt64Arith;
  - GrB_Semiring_new(&UInt64Arith, UInt64Plus, GrB_TIMES_UINT64);
Outline

• Graphs and Linear Algebra
• The GraphBLAS C API and Adjacency Matrices
• GraphBLAS Operations
• Breadth-First Traversal
Breadth First Traversal

• The Breadth First Traversal:
  – Start from one or more initial vertices
  – Visit all accessible one hop neighbors,
  – Visit all accessible unique two hop neighbors,
  – Continue until no more unique vertices to visit
  – Note: keep track of vertices visited so you don’t visit the same vertex more than once

• Breadth first traversal is a common pattern used in a range of graph algorithms
  – Build a spanning tree that contains all vertices and minimal number of edges
  – Search for accessible vertices with certain properties.
  – Find shortest paths between vertices.
  – Other more advanced algorithms such as maxflow and betweenness centrality
Our Breadth First Traversal plan

• We will build up this algorithm using the GraphBLAS through a series of exercises:
  – Wavefronts and how to move from one wavefront to the next.
  – Iteration across wavefronts
  – Track which vertices have been visited
  – Avoid revisiting vertices
  – Construct the Level Breadth first traversal algorithm
Wavefronts

• A subset of vertices accessed at one stage in a breadth first search pattern … for example ….
  – “You tell two friends and they tell two friends…”

Red=current wavefront and visited, Blue=next wavefront, Black=unvisited
Exercise 7: Traverse the graph

• Modify your code from Exercises 5 to iterate from one wavefront to the next.
• Output each wavefront
• How long before you get a repeating pattern?
  - GrB_Vector result, vec;
  - GrB_Index NODE;
  - GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);
  - GrB_Vector_setElement(vec, true, NODE);
  - pretty_print_vector_UINT64(vec, "Input node");
  - GrB_Descriptor desc;
  - GrB_Descriptor_new(&desc);
  - GrB_Descriptor_set(desc, FIELD, VALUE)
  - GrB_mxv(result, GrB_NULL, GrB_NULL, GxB_LOR_LAND_BOOL, graph, vec, desc);
Solution to exercise 7

// First wavefront has one node set.
GrB_Index const SRC_NODE = 0;
GrB_Vector w;
GrB_Vector_new(&w, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(w, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);

pretty_print_vector_UINT64(w,"wavefront(src) ");

for (int i = 0; i < NUM_NODES; ++i) {
    GrB_mxv(w, GrB_NULL, GrB_NULL,
            GxB_LOR_LAND_BOOL, graph, w, desc);
    pretty_print_vector_UINT64(w, "wavefront");
}

The same container can be used for both input and output
Starts repeating after only a few iterations. Why?
Solution to exercise 7: wavefronts

• “We tell a bunch, and they tell bunch…(rinse and repeat)”

Red=current wavefront and visited, Blue=next wavefront, Black=unvisited

w = \{0, 2, 4, 5, 6\}  \quad w = \{1, 2, 3, 4, 5\}  \quad w = \{0, 2, 4, 5, 6\}
Visited lists

• Breadth-first traversal requires that we visit each node once.
• First step is to keep track of a visited list.
• You can do this by accumulating the wavefronts.
  – Use element-wise logical-OR.
Element-wise Operations: Mult and Add

- $\otimes$ assumes unstored values (-) are the binary operator’s *annihilator*:

$$u \otimes v$$

- $\oplus$ assumes unstored values (-) are the binary operator’s *identity*:

$$u \oplus v$$

Examples: $(x,0)$, (and, false), $(+, \infty)$

Examples: $(+,0)$, (or, false), $(\text{min}, \infty)$

The rules for element-wise addition also apply to the accumulation operator, $\odot$
GrB_eWiseMult() \hspace{1cm} w \odot = (u \odot v)

- Compute the element-wise “multiplication” of two GraphBLAS vectors.
- Performs the specified operator (op) on the intersection of the sparse entries in each input vector, u and v.
  - op could be GrB_BinaryOp, GrB_Monoid, or GrB_Semiring
- Returns error codes of type GrB_info. See the spec for details.

GrB_Info GrB_eWiseMult(GrB_Vector w, const GrB_Vector mask, const GrB_BinaryOp accum, const GrB_BinaryOp op, const GrB_Vector u, const GrB_Vector v, const GrB_Descriptor desc); \rightarrow GrB_NULL

Use default values for mask, accum and desc (indicated by GrB_NULL)
GrB_eWiseAdd()  \( \mathbf{w} \ominus = (\mathbf{u} \oplus \mathbf{v}) \)

- Compute the element-wise “addition” of two GraphBLAS vectors.
- Performs the specified operator (op) on the union of the sparse entries in each input vector, \( \mathbf{u} \) and \( \mathbf{v} \).
  - op could be GrB_BinaryOp, GrB_Monoid, or GrB_Semiring
- Returns error codes of type GrB_info. See the spec for details.

GrB_Info GrB_eWiseAdd( GrB_Vector \( \mathbf{w} \),
const GrB_Vector \( \mathbf{mask} \),
const GrB_BinaryOp \( \mathbf{accum} \),
const GrB_BinaryOp \( \mathbf{op} \),
const GrB_Vector \( \mathbf{u} \),
const GrB_Vector \( \mathbf{v} \),
const GrB_Descriptor \( \mathbf{desc} \); )  \( \rightarrow \)  GrB_NULL

Use default values for mask, accum and desc (indicated by GrB_NULL)
Exercise 8: Keep track of ‘visited’ nodes

- Modify code from Exercise 7 to compute the visited set as you iterate.
  - GrB_Vector result, vec;
  - GrB_Index NODE;
  - GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);
  - GrB_Vector_setElement(vec, true, NODE);
  - pretty_print_vector_UINT64(vec, "Input node");
  - GrB_Descriptor desc;
  - GrB_Descriptor_new(&desc);
  - GrB_Descriptor_set(desc, ARG, OP)
    - GrB_eWiseAdd(vec, GrB_NULL, GrB_NULL,
      GrB_LOR, vec, wav, GrB_NULL);
  - GrB_mxv(result, GrB_NULL, GrB_NULL,
    GxB_LOR_LAND_BOOL, graph, vec, desc);
Solution to exercise 8

// First wavefront has node 0 set.
GrB_Index const SRC_NODE = 0;
GrB_Vector w, v;
GrB_Vector_new(&w, GrB_BOOL, NUM_NODES);
GrB_Vector_new(&v, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(w, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);

pretty_print_vector_UINT64(w, "wavefront(src)");

for (int i=0; i<NUM_NODES; ++i) {
    GrB_eWiseAdd(v, GrB_NULL, GrB_NULL,
                   GrB_LOR, v, w, GrB_NULL);
    pretty_print_vector_UINT64(v, "visited");
    GrB_mxv(w, GrB_NULL, GrB_NULL,
            GxB_LOR_LAND_BOOL, graph, w, desc);
    pretty_print_vector_UINT64(w, "wavefront");
}
Solution to exercise 8

// First wavefront has node 0 set.
GrB_Index const SRC_NODE = 0;
GrB_Vector w, v;
GrB_Vector_new(&w, GrB_BOOL, NUM_NODES);
GrB_Vector_new(&v, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(w, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);

pretty_print_vector_UINT64(w, "wavefront(src)");

for (int i=0; i<NUM_NODES; ++i) {
    GrB_eWiseAdd(v, GrB_NULL, GrB_NULL,
                 GrB_LOR, v, w, GrB_NULL);
    pretty_print_vector_UINT64(v, "visited");
    GrB_mxv(w, GrB_NULL, GrB_NULL,
            GxB_LOR_LAND_BOOL, graph, w, tran);
    pretty_print_vector_UINT64(w, "wavefront");
}

What should the exit condition be?

Vector: wavefront(src) =
[ 1, - , - , - , - , - , - ]

Vector: visited =
[ 1, - , - , - , - , - , - ]

Vector: wavefront =
[ - , 1, - , 1, - , - , - ]

Vector: visited =
[ 1, 1, - , 1, - , - , - ]

Vector: wavefront =
[ 1, - , 1, - , 1, - , - ]

Vector: visited =
[ 1, 1, 1, 1, 1, 1, - , 1 ]

Vector: wavefront =
[ - , 1, 1, 1, 1, 1, 1 , - ]

Vector: visited =
[ 1, 1, 1, 1, 1, 1, 1, 1 ]

Vector: wavefront =
[ 1, - , 1, - , 1, 1, 1, 1 ]

Vector: visited =
[ 1, 1, 1, 1, 1, 1, 1, 1 ]

...
It's time to explain masking and REPLACE in GraphBLAS operations.

\[ w \langle \neg m, z \rangle \odot = (A \oplus \cdot \otimes u) \]
Every GraphBLAS operation that computes an opaque matrix or vector supports a “write mask”

A mask, \( m \), controls which elements of the output can be written:
- Same size as output object (mask vectors or mask matrices)
- Any location in the mask that evaluates to ‘true’ can be written in the output object

\[
\mathbf{w}(\langle \mathbf{m} \rangle) = (\mathbf{A} \oplus \cdot \otimes \mathbf{u})
\]

\[\mathbf{w}(\langle \mathbf{m} \rangle) = (\mathbf{A} \oplus \cdot \otimes \mathbf{u})\]
REPLACE vs. “MERGE”

• When a mask is used and the output container is not empty when the operation is called…what do you do to the “masked out” elements?
  – REPLACE (z): all unwritten locations are cleared (zeroed out).
  – MERGE: all unwritten locations are left alone.

• Behaviour defaults to MERGE; otherwise, use a descriptor:
  – GrB_Descriptor_set(desc, GrB_OUTP, GrB_REPLACE)

\[
\text{w}(m, z) = (A \oplus \odot u)
\]
Structural Complement (mask)

• Specified with a descriptor:
  – `GrB_Descriptor_set(desc, GrB_MASK, GrB_SCMP)`
• Inverts the logic of mask (write enabled on false)
• A mask, m, is interpreted as a logical ‘stencil’ that controls which elements of the output can be written:
  – Any location in the mask that evaluates to ‘true’ can be written

\[ w(\neg m, z) = (A \oplus \bullet \otimes u) \]
Using Descriptors (summary)

• A descriptor field selects the object it impacts:
  – GrB_INP0: The first input GraphBLAS object
  – GrB_INP1: The second input GraphBLAS object
  – GrB_MASK: The GraphBLAS mask object
  – GrB_OUTP: The output GraphBLAS object

• Each field supports one value (currently):
  – GrB_INP0: GrB_TRAN (transpose)
  – GrB_INP1: GrB_TRAN (transpose)
  – GrB_MASK: GrB_SCMP (structural complement)
  – GrB_OUTP: GrB_REPLACE (clear the output before writing result)
Exercise 9: Avoid revisiting

- Use the visited list as a mask prevent revisiting previous nodes
- Exit the loop when there is no more ‘work’ to be done
- You will need the following types and methods from the GraphBLAS
  - `GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);`
  - `GrB_Vector_setElement(vec, true, NODE);`
  - `GrB_eWiseAdd(vec, GrB_NULL, GrB_NULL, GrB_LOR, vec, wav, GrB_NULL);`
  - `GrB_mxv(result, GrB_NULL, GrB_NULL, GxB_LOR_LAND_BOOL, graph, vec, desc);`
  - `GrB_Descriptor desc;`
  - `GrB_Descriptor_new(&desc);`
  - `GrB_Descriptor_set(desc, FIELD, VALUE)`

FIELD: `GrB_INPO, GrB_INP1, GrB_OUTP, GrB_MASK`
VALUE: `GrB_TRAN, GrB_REPLACE, GrB_SCMP`
Solution to exercise 9

... 
GrB_Vector_setElement(w, true, SRC_NODE);

GrB_Descriptor desc;
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);

pretty_print_vector_UINT64(w, "wavefront(src)");

GrB_Index nvals = 0;
do {
    GrB_eWiseAdd(v, GrB_NULL, GrB_NULL,
        GrB_LOR, v, w, GrB_NULL);
    pretty_print_vector_UINT64(v, "visited");
    GrB_mxv(w, v, GrB_NULL,
        GxB_LOR_LAND_BOOL, graph, w, desc);
    pretty_print_vector_UINT64(w, "wavefront");
    GrB_Vector_nvals(&nvals, w);
} while (nvals > 0);
Breadth-First Traversal

Red=current wavefront and visited, Blue=next wavefront, Gray=visited, Black=unvisited
How would we keep track of when each vertex is visited (becomes red)?
GrB_assign() from constant \( w(i) \odot = c \)

- Assign a constant to a subset of the output vector.
- Locations to be assigned selected by an output index vector, indices:

\[
\begin{align*}
\text{w}(\text{indices}[j]) &= c, & \forall j : 0 \leq j < \text{nindices}, \\
\text{w}(\text{indices}[j]) &= \text{w}(\text{indices}[j]) \odot c, & \forall j : 0 \leq j < \text{nindices}.
\end{align*}
\]

GrB_Info GrB_assign(  GrB_Vector w,  const GrB_Vector mask,  const GrB_BinaryOp accum,  const GrB_Vector u,  const GrB_Index* indices,  const GrB_Index nindices,  const GrB_Descriptor desc);

- Use a constant GrB_ALL in place of the indices argument to select that all elements of \( w \) are to be assigned to (in order 0 to 1-nindices).
There are several variants of assign

- Standard vector assignment
- Standard matrix assignment

\[ \mathbf{w}(i) \odot = \mathbf{u} \quad \mathbf{C}(i, j) \odot = \mathbf{A} \]

- Assign a vector to the elements of column \( c_j \) of a matrix
- Assign a vector to the elements of row \( r_i \) of a matrix

\[ \mathbf{C}(i, c_j) \odot = \mathbf{u} \quad \mathbf{C}(r_i, j) \odot = \mathbf{u}^\top \]

- Assign a constant to a subset of a vector.
- Assign a constant to a subset of a matrix.

\[ \mathbf{w}(i) \odot = c \quad \mathbf{C}(i, j) \odot = c \]

\( \mathbf{A} \) and \( \mathbf{C} \) are GraphBLAS matrices. \( \mathbf{u} \) and \( \mathbf{w} \) are GraphBLAS vectors \( \mathbf{i} \) and \( \mathbf{j} \) are index vectors.
GrB_assign() from vector  \( w(i) \odot = u \)

- Assign a vector to a subset of the output vector.
- Values to be assigned selected by an output index vector, \( i \)

\[
\begin{align*}
    w(\text{indices}[j]) &= u(j), & \forall \ j : 0 \leq j < n\text{indices}, \\
    w(\text{indices}[j]) &= w(\text{indices}[j]) \odot u(j), & \forall \ j : 0 \leq j < n\text{indices}.
\end{align*}
\]

GrB_Info GrB_assign(
    GrB_Vector w,
    const GrB_Vector mask,
    const GrB_BinaryOp accum,
    const GrB_Vector u,
    const GrB_Index *indices,
    const GrB_Index nindices,
    const GrB_Descriptor desc);

- Use a constant \texttt{GrB\_ALL} in place of the \texttt{indices} argument to select that all elements of \( u \) are to be assigned in order to \( w \).
Exercise 10: level BFS

• Modify the code from Exercise 9 to compute the level at which each node is encountered:
  – SRC_NODE is level 1, Its neighbors are level 2, ... and so forth

• Challenge: use assign in place of eWiseAdd

- `GrB_Vector_new(&w, GrB_BOOLEAN, NUM_NODES);`
- `GrB_Vector_setElement(w, true, SRC_NODE);`
- `GrB_Descriptor desc;`
- `GrB_Descriptor_new(&desc);`
- `GrB_Descriptor_set(desc, FIELD, VALUE);`
- `pretty_print_vector_UINT64(vec, "levels");`
- `GrB_assign(u, mask, accum, c, GrB_ALL, NUM_NODES, desc);`
- `GrB_mxv(w, mask, accum, GxB_LOR_LAND_BOOLEAN, graph, w, desc);`
- `GrB_Vector_nvals(&nvals, w);`

FIELD: GrB_INP0, GrB_INP1, GrB_OUTP, GrB_MASK
VALUE: GrB_TRAN, GrB_REPLACE, GrB_SCMP
Solution to exercise 10

GrB_Vector_new(&levels, GrB_UINT64, NUM_NODES);
GrB_Vector_new(&w, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(w, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, ...);

pretty_print_vector_BOOL(w, "wavefront(src)");
GrB_Index nvals = 0, lvl = 0;
do {
    ++lvl;
    GrB_assign(levels, w, GrB_NULL,
               lvl, GrB_ALL, NUM_NODES, GrB_NULL);
    pretty_print_vector_UINT64(levels, "levels");

    GrB_mxv(w, levels, GrB_NULL,
            GxB_LOR_LAND_BOOL, graph, w, desc);
    pretty_print_vector_BOOL(w, "wavefront");
    GrB_Vector_nvals(&nvals, w);
} while (nvals > 0);
The GraphBLAS Operations

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Mathematical Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{m}x\text{m}</td>
<td>\text{C} \langle \text{M}, z \rangle = \text{C} \odot \text{A} \oplus \odot \text{B}</td>
</tr>
<tr>
<td>\text{m}x\text{v}</td>
<td>\text{w} \langle \text{m}, z \rangle = \text{w} \odot \text{A} \oplus \odot \text{u}</td>
</tr>
<tr>
<td>\text{v}x\text{m}</td>
<td>\text{w}^T \langle \text{m}^T, z \rangle = \text{w}^T \odot \text{u}^T \oplus \odot \text{A}</td>
</tr>
<tr>
<td>\text{eWiseMult}</td>
<td>\text{C} \langle \text{M}, z \rangle = \text{C} \odot \text{A} \otimes \text{B}</td>
</tr>
<tr>
<td>\text{eWiseAdd}</td>
<td>\text{w} \langle \text{m}, z \rangle = \text{w} \odot \text{u} \oplus \text{v}</td>
</tr>
<tr>
<td>\text{reduce} (row)</td>
<td>\text{w} \langle \text{m}, z \rangle = \text{w} \odot [\oplus_j \text{A}(i, j)]</td>
</tr>
<tr>
<td>\text{reduce} (scalar)</td>
<td>s = s \odot [\oplus_{i,j} \text{A}(i, j)]</td>
</tr>
<tr>
<td>\text{apply}</td>
<td>\text{C} \langle \text{M}, z \rangle = \text{C} \odot f_u(\text{A})</td>
</tr>
<tr>
<td>\text{transpose}</td>
<td>\text{C} \langle \text{M}, z \rangle = \text{C} \odot \text{A}^T</td>
</tr>
<tr>
<td>\text{extract}</td>
<td>\text{C} \langle \text{M}, z \rangle = \text{C} \odot \text{A}(i, j)</td>
</tr>
<tr>
<td>\text{assign}</td>
<td>\text{C} \langle \text{M}, z \rangle(i, j) = \text{C}(i, j) \odot \text{A}</td>
</tr>
</tbody>
</table>

The same conventions are used across all operations so the operations we did not cover are straightforward to pick up.
Conclusion and next steps

• The GraphBLAS define a standard API for “Graph Algorithms in the Language of Linear Algebra”.

• A wide range of algorithms are variations of the basic breadth first traversal for a graph.

• To reach GraphBLAS mastery
  – Attend the Graph Algorithms Building Blocks workshop at IPDPS
  – Explore the challenge problems included with this tutorial
  – Work through the algorithms in the Graph book →
GraphBLAS at HPEC 2018

• GraphBLAS is a community effort. Join the community:
  – Go to graphblas.org and join our mailing list

• Attend the HPEC GraphBLAS Birds of a Feather (BOF) 6 PM to 7 PM, Eden Vale C1/C2.

• Please send us feedback about the tutorial
  timothy.g.mattson@intel.com
  smcmillan@sei.cmu.edu
  – Tell us what you really liked.
  – Tell us what we should change
  – Tell us what you wish we’d covered but didn’t
  – Plus anything else that might help us improve
Appendices

• MxM: the low-level details of the GraphBLAS operations
• Challenge Problems: Some key algorithms with the GraphBLAS
• SuiteSparse: usage notes, extensions and future plans
• Reference material
GraphBLAS: details of operations

- When you read the GraphBLAS C API specification, the operations are described in a manner that may seem obtuse.

- The definitions, however, are presented in this way for good reasons:
  - to cover the full range of variations exposed by the various arguments and to express the operation without ever specifying the undefined elements (i.e. the “zeros” of the semiring).
  - To avoid any reference to the non-stored elements of the sparse matrix. In sparse arrays, the undefined elements are usually assumed to be the ”zero of the semiring”. By defining the operations without any reference to those “un-stored values”, we can freely change the semirings between operations without having to update the un-stored elements.
GrB_mxm()

\[ C = A \oplus \otimes B = AB \]

Matrix Multiplication ... the way we learned it in school

\[ C(i, j) = \bigoplus_{k=1}^{l} A(i, k) \otimes B(k, j) \]

\[ A : S^{m \times l} \quad B : S^{l \times n} \quad C : S^{m \times n} \]

Matrix Multiplication ... set notation to ignore un-stored elements

\[ C(i, j) = \bigoplus_{k \in \text{ind}(A(i,:)) \cap \text{ind}(B(:,j))} (A(i, k) \otimes B(k, j)) \]

With set notation, it’s easier to define the operations over a matrix as the semi-ring changes
GrB_mxm(): Function Signature

GrB_Info GrB_mxm(GrB_Matrix *C,
    const GrB_Matrix Mask,
    const GrB_BinaryOp accum,
    const GrB_Semiring op,
    const GrB_Matrix A,
    const GrB_Matrix B,
    const GrB_Descriptor desc);

C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the matrix product. On output, the matrix holds the results of this operation.

Mask (IN) A “write” mask that controls which results from this operation are stored into the output matrix C (optional). If no mask is desired, GrB_NULL is specified. The Mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any “built-in” GraphBLAS type.

accum (IN) A binary operator used for accumulating entries into existing C entries. For assignment rather than accumulation, GrB_NULL is specified.

op (IN) Semiring used in the matrix-matrix multiply: \( op = \langle D_1, D_2, D_3, \oplus, \otimes, 0 \rangle \).

A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the multiplication.

B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the multiplication.

desc (IN) Operation descriptor (optional). If a default descriptor is desired, GrB_NULL should be used. Valid fields are as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GrB_OUTP</td>
<td>GrB_REPLACE</td>
<td>Output matrix C is cleared (all elements removed) before result is stored in it.</td>
</tr>
<tr>
<td>Mask</td>
<td>GrB_MASK</td>
<td>GrB_SCMP</td>
<td>Use the structural complement of Mask.</td>
</tr>
<tr>
<td>A</td>
<td>GrB_INP0</td>
<td>GrB_TRAN</td>
<td>Use transpose of A for operation.</td>
</tr>
<tr>
<td>B</td>
<td>GrB_INP1</td>
<td>GrB_TRAN</td>
<td>Use transpose of B for operation.</td>
</tr>
</tbody>
</table>
GrB_mxm(): Function Signature

GrB_Info GrB_mxm(GrB_Matrix *C,
const GrB_Matrix Mask,
const GrB_BinaryOp accum,
const GrB_Semiring op,
const GrB_Matrix A,
const GrB_Matrix B,
const GrB_Descriptor desc);

GrB_Info return values:

GrB_SUCCESS
Blocking mode: Operations completed successfully.
Nonblocking mode: consistency tests passed on dimensions and domains for input arguments

GrB_PANIC
Unknown Internal error

GrB_OUTOFMEM
Not enough memory for the operation

GrB_DIMENSION_MISMATCH
Matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH
Domains of matrices are incompatible with the domains of the accumulator, semiring, or mask.
## Standard function behavior

- Consider the following code:

  ```c
  GrB_Descriptor_new(&desc);
  GrB_Descriptor_set(desc, GrB_OUTP, GrB_REPLACE);
  GrB_Descriptor_set(desc, GrB_INP0, GrB_TRANS);
  GrB_mxm(&C, M, Int32Add, Int32AddMul, A, B, desc);
  ```

<table>
<thead>
<tr>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form input operands and mask based on descriptor</td>
<td>C, B, M, A ← A&lt;sup&gt;T&lt;/sup&gt;</td>
</tr>
<tr>
<td>Test the domains and sizes for consistency.</td>
<td>int32, dims match</td>
</tr>
<tr>
<td>Carry out the indicated operation</td>
<td>T ← A * .+ B, Z ← C + T</td>
</tr>
<tr>
<td>Apply the write-mask to select output values</td>
<td>Z ← Z ∩ M</td>
</tr>
<tr>
<td>Replace mode: delete elements in output object and replace with output values</td>
<td>C ← Z</td>
</tr>
<tr>
<td>Merge mode: Assign output value (i,j) to element (i,j) of output object, but leave other elements of the output object alone.</td>
<td></td>
</tr>
</tbody>
</table>

int32AddMul semiring
int32Add accumulation
To understand what happens inside a graphBLAS operation, consider matrix multiply.

All the operations follow this basic format

```c
GrB_Info GrB_mxm(
    GrB_Matrix C,
    const GrB_Matrix M,
    const GrB_BinaryOp accum,
    const GrB_Semiring op,
    const GrB_Matrix A,
    const GrB_Matrix B,
    const GrB_Descriptor desc);
```

GrB_Info GrB_mxm takes six arguments:
- `C`: The result matrix.
- `M`: One of the input matrices.
- `accum`: The accumulation operation.
- `op`: The binary operation.
- `A`: Another input matrix.
- `B`: Another input matrix.
- `desc`: A descriptor for the operation.

The function returns a `GrB_Info` object that can be used to check the success of the operation or to retrieve error information.
Exercise: Matrix Matrix Multiplication

- Multiply the adjacency matrix from our “logo graph” by itself.
- Print resulting matrix and interpret the result.
- Hint: Do the multiply again and compare results. Do you see the pattern?
Appendices

• MxM: the low-level details of the GraphBLAS operations
• Challenge Problems: Some key algorithms with the GraphBLAS
• SuiteSparse: usage notes, extensions and future plans
• Reference material
Challenge problems

- Triangle counting
- PageRank
- Betweenness Centrality
- Maximal Independent Set

Work in Progress: We should make a slide for each problem defining the algorithm in enough detail so students can implement the GraphBLAS implementation on their own.
Counting Triangles (once) with GraphBLAS

- Given:
  - Undirected graph $G = \{V, E\}$
  - $L$: boolean, lower-triangular portion of adjacency matrix
- # triangles $= ||L \otimes (L \oplus \otimes L^T)||_1$
  - Semiring can be Plus-AND or Plus-Times
  - Element-wise multiplication is equivalent to a mask operation

```c
uint64_t triangle_count(GrB_Matrix L) // L: Nxn, lower-triangular, boolean
{
    GrB_Index N;
    GrB_Matrix_nrows(&N, L);
    GrB_matrix C;
    GrB_Matrix_new(&C, GrB_UINT64, N, N);

    GrB_mxm(C, L, GrB_NULL, GrB_UINT64AddMul, L, L, GrB_TB); // C<L> = L * L^T

    uint64_t count;
    GrB_reduce(&count, GrB_NULL, GrB_UINT64Add, C, GrB_NULL); // 1-norm of C
    return count;
}
```
Appendices

- MxM: the low-level details of the GraphBLAS operations
- Challenge Problems: Some key algorithms with the GraphBLAS
- SuiteSparse: usage notes, extensions and future plans
- Reference material
SuiteSparse:GraphBLAS

• Full implementation of GraphBLAS Specification written by Tim Davis, Texas A&M University
• Easy-to-read User Guide with lots of examples
• Already in Ubuntu, Debian, Mac HomeBrew, ...
• Most operations just as fast as MATLAB (like C=A*B)
• assign and setElement can be 1000x faster (or more!) than MATLAB, by exploiting non-blocking mode
• V2.1: matrices by-row and by-column; by-row is often faster than by-column when A(i,j) is the edge (i,j). Compile with –DBYROW or use GxB_set(…)
• Graph algorithms in GraphBLAS typically faster than novice-level graph algorithm without GraphBLAS, and easier to write

• http://faculty.cse.tamu.edu/davis/GraphBLAS
SuiteSparse:GraphBLAS extensions

- MATLAB-like colon notation for GrB_assign, extract
- unary operators ONE, ABS, LNOT_[type]
- ISEQ, ISNE, ISLT, ... return same type as inputs (e.g. PLUS monoid cannot be combined with Boolean EQ, but PLUS-ISEQ can, to count the number of equal pairs)
- query: size of type, type of matrix, ...
- GxB_select: like MATLAB L=tril(A,k), d=diag(A), ...
- GxB_get/set: to change matrix format (by row, by col, hypersparse)
- 44 built-in monoids
- 960 built-in semirings (like GxB_LOR_LAND_BOOL)
- GxB_resize: change size of matrix or vector
- GxB_subassign: variation of GrB_assign
- GxB_kron: Kronecker product
- Thread-safe if called by user application threads, in parallel
SuiteSparse:GraphBLAS future

- Multicore parallelism via OpenMP
- Variable-sized types (imagine matrix of matrices, or a matrix of arbitrary-sized integers with 10’s or 1000’s of digits)
- Solvers: $Ax=b$ over a group (double, GF(2), ...)
- Better performance: e.g. many monoids could terminate quickly:
  - OR ($x_1, x_2, x_3, ...$) becomes true as soon as any $x_i = true$
  - also for AND, and reduction ops FIRST and SECOND
- Iterators for algorithms like depth-first-search
- Reduction to vector or scalar: could also return the index for some operators (MAX, MIN, FIRST, SECOND): argmin, argmax
- Pretty-print methods
- Serialization to/from a binary string: for binary file I/O, or sending/receiving a GrB_Matrix in an MPI message; with compression
- Priority queue: a GrB_Vector acting like a heap
- Concatenate: like $C=[A;B]$ in MATLAB
- Interface to MATLAB, Julia, Python, ...
- Faster $C=A*B$ for user-defined types and operators

Content provided by Tim Davis
Appendices

• MxM: the low-level details of the GraphBLAS operations
• Challenge Problems: Some key algorithms with the GraphBLAS
• SuiteSparse: usage notes, extensions and future plans
• Reference material
Full set of GraphBLAS opaque objects

<table>
<thead>
<tr>
<th>GrB_Object types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_Type</td>
<td>User-defined scalar type.</td>
</tr>
<tr>
<td>GrB_UnaryOp</td>
<td>Unary operator, built-in or associated with a single-argument C function.</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>Binary operator, built-in or associated with a two-argument C function.</td>
</tr>
<tr>
<td>GrB_Monoid</td>
<td>Monoid algebraic structure.</td>
</tr>
<tr>
<td>GrB_Semiring</td>
<td>A GraphBLAS semiring algebraic structure.</td>
</tr>
<tr>
<td>GrB_Matrix</td>
<td>Two-dimensional collection of elements; typically sparse.</td>
</tr>
<tr>
<td>GrB_Vector</td>
<td>One-dimensional collection of elements.</td>
</tr>
<tr>
<td>GrB_Descriptor</td>
<td>Descriptor object, used to modify behavior of methods.</td>
</tr>
</tbody>
</table>
## Error codes returned by GraphBLAS methods

### API Errors

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_UNINITIALIZED_OBJECT</td>
<td>A GraphBLAS object is passed to a method before new was called on it.</td>
</tr>
<tr>
<td>GrB_NULL_POINTER</td>
<td>A NULL is passed for a pointer parameter.</td>
</tr>
<tr>
<td>GrB_INVALID_VALUE</td>
<td>Miscellaneous incorrect values.</td>
</tr>
<tr>
<td>GrB_INVALID_INDEX</td>
<td>Indices passed are larger than dimensions of the matrix or vector being accessed.</td>
</tr>
<tr>
<td>GrB_DOMAIN_MISMATCH</td>
<td>A mismatch between domains of collections and operations when user-defined domains are in use.</td>
</tr>
<tr>
<td>GrB_DIMENSION_MISMATCH</td>
<td>Operations on matrices and vectors with incompatible dimensions.</td>
</tr>
<tr>
<td>GrB_OUTPUT_NOT_EMPTY</td>
<td>An attempt was made to build a matrix or vector using an output object that already contains valid tuples (elements).</td>
</tr>
<tr>
<td>GrB_NO_VALUE</td>
<td>A location in a matrix or vector is being accessed that has no stored value at the specified location.</td>
</tr>
</tbody>
</table>
### Error codes returned by GraphBLAS methods

#### Execution Errors

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB.OUT_OF_MEMORY</td>
<td>Not enough memory for operations. The array provided is not large enough to hold output.</td>
</tr>
<tr>
<td>GrB.INSUFFICIENT_SPACE</td>
<td>One of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error.</td>
</tr>
<tr>
<td>GrB.INVALID_OBJECT</td>
<td>Reference to a vector or matrix element that is outside the defined dimensions of the object. Unknown internal error.</td>
</tr>
<tr>
<td>GrB.INDEX_OUT_OF_BOUNDS</td>
<td></td>
</tr>
<tr>
<td>GrB.PANIC</td>
<td></td>
</tr>
</tbody>
</table>