GraphBLAS:
Building a C++ Matrix API for Graph Algorithms

BENJAMIN BROCK, SCOTT MCMILLAN
About Us

**Ben**, PhD Candidate at UC Berkeley

Data structures and algorithms for **parallel programs**. Working on C++ library of **distributed data structures**. **Please hire me!**

**Scott**, Principal Engineer at CMU SEI

**Graph/ML/AI algorithms** for large- and small-scale parallel systems. Working on **GBTL**, a linear algebra-based C++ library for graph analytics.
Copyright 2021 Carnegie Mellon University and Benjamin Brock.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

DM21-0916
This Talk

Background: How and why to use matrix algebra for graphs?

What are the important data structures and concepts?

Prior work in the GraphBLAS community, C API

Overview of our draft C++ API

How might this interoperate with standard C++, graph library proposal?
This Talk

**Background:** How and why to use matrix algebra for graphs?

What are the important data structures and concepts?

Prior work in the GraphBLAS community, C API

Overview of our draft C++ API

How might this interoperate with standard C++, graph library proposal?
This Talk

**Background:** How and why to use matrix algebra for graphs? What are the important data structures and concepts?

Prior work in the GraphBLAS community, C API

Overview of our draft C++ API

How might this interoperate with standard C++, graph library proposal?
This Talk

**Background:** How and why to use matrix algebra for graphs?

What are the important **data structures** and **concepts**?

Prior work in the **GraphBLAS community**, C API

Overview of our **draft C++ API**

How might this interoperate with **standard C++**, **graph library** proposal?
This Talk

Background: How and why to use matrix algebra for graphs?

What are the important data structures and concepts?

Prior work in the GraphBLAS community, C API

Overview of our draft C++ API

How might this interoperate with standard C++, graph library proposal?
What This Talk Is Not

- A C++ standards proposal

- A complete evaluation of graph programming models
What This Talk Is Not

- A C++ standards proposal

- A complete evaluation of graph programming models
Background: How and why to use matrix algebra for graphs
Graphs: Understanding relationships between items

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

Graph is a pair \((V, E)\):
- \(V\) is a set of vertices
- \(E\) is a set of paired vertices (edges)

\[V = \{0, 1, 2, 3, 4, 5, 6\}\]
\[E = \{(0,1), (0,3), (1,4), (1,6), (2,5), (3,0), (3,2), (4,5), (5,2), (6,2), (6,3), (6,4)\}\]

Ordered pairs results in directed graphs (shown)
Graph Analysis is Important and Pervasive

**Social**
- Graphs represent relationships between individuals or documents
- 100,000s – 100,000,000s individuals and interactions

**Cyber**
- Graphs represent communication patterns of computers on a network
- 1,000,000s – 1,000,000,000s network events

**Biology**
- Graphs represent organization of neural interactions within the brain
- $10^{11} – 10^{15}$ neurons and connections
Graphs as Adjacency Matrices

Graphs are represented as adjacency matrices that usually have *sparse* and *irregular* structure.

\[
A_{ij} = \begin{cases} 
\bullet & (v_i, v_j) \in E \\
\emptyset & (v_i, v_j) \notin E 
\end{cases}
\]
GraphBLAS Timeline

Book — Papers — GraphBLAS standards — SuiteSparse:GraphBLAS releases


Graph Algorithms in the Language of Linear Algebra

Standards for graph algorithm primitives, HPEC

Seven good reasons, ICCS

Mathematical foundations, HPEC

C API, GABB@IPDPS

LAGraph, GrAPL@IPDPS

C++ API Roadmap, GrAPL@IPDPS

GraphBLAS Timeline

0.9 1.0 1.2 1.3 2.0 2.2 3.0 4.0 5.1
The GraphBLAS “standard”

**Goal:** separate the concerns of the hardware/library/application designers.

1979: BLAS Basic Linear Algebra Subprograms (BLAS 2 ’88, BLAS 3 ’90)

Numerical applications

LINPACK/LAPACK

API: Separation of concerns

BLAS

Hardware architecture
The GraphBLAS “standard”

**Goal:** separate the concerns of the hardware/library/application designers.

1979: BLAS Basic Linear Algebra Subprograms (BLAS 2 ’88, BLAS 3 ’90)
2001: Sparse BLAS an extension to BLAS (little uptake)

![Diagram showing API separation of concerns]

- Numerical applications
- LINPACK/LAPACK
- API: Separation of concerns
- BLAS
- Hardware architecture
The GraphBLAS “standard”

**Goal:** separate the concerns of the hardware/library/application designers.

- **1979: BLAS** Basic Linear Algebra Subprograms (BLAS 2 ’88, BLAS 3 ’90)
- **2001: Sparse BLAS** an extension to BLAS (little uptake)
- **2013: GraphBLAS** an effort to define standard building blocks for graph algorithms in the language of linear algebra
Graphs as Adjacency Matrices

\[ A_{ij} = \begin{cases} 
\bullet & (v_i, v_j) \in E \\
\emptyset & (v_i, v_j) \notin E
\end{cases} \]
Graphs as Adjacency Matrices

\[ A_{ij} = \begin{cases} \bullet & (v_i, v_j) \in E \\ \emptyset & (v_i, v_j) \notin E \end{cases} \]
Graphs as Adjacency Matrices

\[ A_{ij} = \begin{cases} \bullet & (v_i, v_j) \in E \\ \emptyset & (v_i, v_j) \notin E \end{cases} \]
Graph Operations as Matrix Operations

- Matrix-vector multiply → find neighbors
  - In-neighbors: use A
  - Out-neighbors: use $A^T$
Graph Operations as Matrix Operations

Finding out-neighbors is used many graph algorithms.

- Matrix-vector multiply → find neighbors
  - In-neighbors: use \( A \)
  - Out-neighbors: use \( A^T \)
Graph Operations as Matrix Operations

Another way to look at matrix-vector multiply…
What is $\bigoplus \times$ ??
Matrix multiplication

Conventional matrix multiplication uses arithmetic plus (+) and times (x):

\[ y = Ax \]

\[ y(i) = \sum_k A(i, k) \cdot x(k) \]
Matrix multiplication on semirings

Conventional matrix multiplication uses arithmetic plus (+) and times (x):

\[ y = A \cdot x \]

\[ y(i) = \sum_k A(i, k) \cdot x(k) \]

The generalized form uses “arbitrary” operators “plus” (\(\oplus\)) and “times” (\(\otimes\)):

\[ y = A \oplus \otimes x \]

\[ y(i) = \bigoplus_k A(i, k) \otimes x(k) \]
Matrix multiplication on semirings

Conventional matrix multiplication uses arithmetic plus (+) and times (x):

\[ y = A \cdot x \]
\[ y(i) = \sum_k A(i, k) \cdot x(k) \]

The generalized form uses “arbitrary” operators “plus” (⊕) and “times” (⊗):

\[ y = A \oplus \otimes x \]
\[ y(i) = \bigoplus_k A(i, k) \otimes x(k) \]

A cornerstone of GraphBLAS: Supports arbitrary semirings that override the addition and multiplication operators (⊕, ⊗).
GraphBLAS semirings $\oplus, \otimes$

- $\oplus$ is commutative binary operator with an identity, $0$ (called a monoid)
- $\otimes$ is a binary operator.
- The identity of $\oplus$, is the annihilator of $\otimes^*$
  - $a = a \oplus 0 = 0 \oplus a$
  - $0 = a \otimes 0 = 0 \otimes a$

<table>
<thead>
<tr>
<th>Semiring</th>
<th>Valid values</th>
<th>$\oplus$</th>
<th>$\otimes$</th>
<th>0</th>
<th>Graph semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer arithmetic</td>
<td>$a \in \mathbb{N}$</td>
<td>$+$</td>
<td>$\cdot$</td>
<td>$0$</td>
<td>number of paths</td>
</tr>
<tr>
<td>real arithmetic</td>
<td>$a \in \mathbb{R}$</td>
<td>$+$</td>
<td>$\cdot$</td>
<td>$0$</td>
<td>strength of all paths</td>
</tr>
<tr>
<td>boolean</td>
<td>$a \in {\text{false, true}}$</td>
<td>$\lor$</td>
<td>$\land$</td>
<td>false</td>
<td>connectivity</td>
</tr>
<tr>
<td>min-plus (tropical)</td>
<td>$a \in \mathbb{R} \cup {+\infty}$</td>
<td>$\min$</td>
<td>$+$</td>
<td>$+\infty$</td>
<td>shortest path</td>
</tr>
<tr>
<td>max-plus</td>
<td>$a \in \mathbb{R} \cup {-\infty}$</td>
<td>$\max$</td>
<td>$+$</td>
<td>$-\infty$</td>
<td>longest path</td>
</tr>
</tbody>
</table>

*In GraphBLAS this is not enforced nor required*
GraphBLAS Primitives

- Basic objects (opaque types)
  - Matrices (sparse or dense), vectors (sparse or dense), algebraic operators (semirings)
- Fundamental operations over these objects

...plus reduction, transpose, Kronecker product, filtering, transform, etc.
One more thing... write masks: $\langle m \rangle$

Often not interested in some nodes...
One more thing... write masks: $\langle m \rangle$

Often not interested in some nodes...

ANOTHER feature of GraphBLAS:
All operations support a write mask.

$$f'\langle m \rangle = A^T \oplus \otimes f$$
Example: Breadth-First Search (levels)

\[ f(src) = \bullet \]
Example: Breadth-First Search (levels)

\[ level = 0 \]
\[ v += level \times f \]
Example: Breadth-First Search (levels)

level = 0
\[ \mathbf{v} += level \times \mathbf{f} \]  // Use \( \mathbf{v} \) as a mask, \( \langle \mathbf{v} \rangle \).
Example: Breadth-First Search (levels)

\[ \text{level} = 0 \]
\[ \mathbf{v} += \text{level} \times \mathbf{f} \]
\[ f'\langle \bar{\mathbf{v}} \rangle = \mathbf{A}^T \oplus \otimes \mathbf{f} \]

// Boolean semiring
Example: Breadth-First Search (levels)

\[ level = 0 \]
\[ v += \text{level} \times f \]
\[ f'(\bar{v}) = A^T \bigoplus \otimes f \]
\[ f = f' \]
Example: Breadth-First Search (levels)

\[ \text{level} = 1 \]
\[ \mathbf{v} += \text{level} \times \mathbf{f} \]
\[ f' \langle \mathbf{v} \rangle = \mathbf{A}^\top \odot \otimes \mathbf{f} \]
\[ \mathbf{f} = f' \]
Example: Breadth-First Search (levels)

\[ level = 2 \]
\[ \mathbf{v} += level \ast \mathbf{f} \]
\[ \mathbf{f}'(\mathbf{v}) = \mathbf{A}^T \odot \otimes \mathbf{f} \]
\[ \mathbf{f} = \mathbf{f}' \]
Example: Breadth-First Search (levels)

\[ \text{level} = 3 \]
\[ \mathbf{v} ++= \text{level} \times \mathbf{f} \]
\[ \mathbf{f}'(\bar{\mathbf{v}}) = \mathbf{A}^T \oplus \bigotimes \mathbf{f} \]
\[ \mathbf{f} = \mathbf{f}' \]
if \( \mathbf{f}.empty() \) return \( \mathbf{v} \)
Example: Breadth-First Search (levels)

- Input: adjacency matrix $A$ (Boolean), source vertex $src$ (integer)
- Output: visited vertices vector, $v$ (integer)
- Workspace: frontier vector $f$ (Boolean)

1. $f(src) = true$
2. $level = 0$
3. while $!f$.empty()
4. $v += level \times f$
5. $f(v) = A^T \oplus \otimes f$  // using the Boolean semiring (OR.AND)
6. $++level$
Prior work: GraphBLAS C API and Onwards
GraphBLAS C API

- Provides **uniform API** for **graph algorithms** in the **language of linear algebra**

- Revolve around sparse matrix and vector operations which can use **arbitrary semirings** instead of classical (+, *)

- Current version of C API spec. is 1.3 (**2.0 arriving imminently!**)  

- C offers great **portability** (Python, bindings, etc.), but has some **disadvantages**...
GraphBLAS C API

- Provides **uniform API** for graph algorithms in the **language of linear algebra**

- Revolve around sparse matrix and vector operations which can use **arbitrary semirings** instead of classical (+, *)

- Current version of C API spec. is 1.3 (**2.0 arriving imminently!**) 

- C offers great **portability** (Python, bindings, etc.), but has some **disadvantages**...
GraphBLAS C API

- Provides uniform API for graph algorithms in the language of linear algebra

- Revolve around sparse matrix and vector operations which can use arbitrary semirings instead of classical (+, *)

- Current version of C API spec. is 1.3 (**2.0 arriving imminently!**)

- C offers great portability (Python, bindings, etc.), but has some disadvantages...
GraphBLAS C API

- Provides **uniform API** for graph algorithms in the **language of linear algebra**

- Revolve around sparse matrix and vector operations which can use **arbitrary semirings** instead of classical (+, *)

- Current version of C API spec. is 1.3 (**2.0 arriving imminently!**) 

- C offers great **portability** (Python, bindings, etc.), but has some **disadvantages**...
The Problem with Types...

- If you’re familiar with the (C)BLAS, there is a function for each scalar type

- GraphBLAS supports a wide variety of scalar types and binary operators

- Combinatorial explosion

\[
\begin{align*}
\text{float* } & \text{ a_ptr = get_matrix(...); } \\
\text{cblas_sgemm(..., m, n, k, 1.0f, a_ptr, ...); } \\
\text{...} \\
\text{double* } & \text{ a_ptr = get_matrix(...); } \\
\text{cblas_dgemm(..., m, n, k, 1.0, a_ptr, ...); }
\end{align*}
\]
The Problem with Types...

- If you’re familiar with the (C)BLAS, there is a function for each scalar type

- GraphBLAS supports a wide variety of scalar types and binary operators

- Combinatorial explosion

```c
float* a_ptr = get_matrix(...);
cblas_sgemm(..., m, n, k, 1.0f, a_ptr, ...);
...

double* a_ptr = get_matrix(...);
cblas_dgemm(..., m, n, k, 1.0, a_ptr, ...);
```
The Problem with Types...

- If you’re familiar with the (C)BLAS, there is a function for each scalar type

- GraphBLAS supports a wide variety of scalar types and binary operators

- Combinatorial explosion

```c
float* a_ptr = get_matrix(...);
cblas_sgemm(..., m, n, k, 1.0f, a_ptr, ...);
...

double* a_ptr = get_matrix(...);
cblas_dgemm(..., m, n, k, 1.0, a_ptr, ...);
```
C API: Quality of Life Issues

- For each predefined GraphBLAS operator, the C API requires a separate C function for each of 11 predefined types:

  GrB_PLUS_BOOL, GrB_PLUS_INT8, GrB_PLUS_UINT8, GrB_PLUS_INT16, GrB_PLUS_UINT16, GrB_PLUS_INT32,
  GrB_PLUS_UINT32, GrB_PLUS_INT64, GrB_PLUS_UINT64, GrB_PLUS_FP32, GrB_PLUS_FP64.

- There are over 1000 combinations of predefined operators and types.

- Creates a large burden on implementers, who mostly resort to automatic code generation
C API: Quality of Life Issues

- For each predefined **GraphBLAS operator**, the C API requires a separate **C function** for each of 11 predefined types:
  
  GrB_PLUS_BOOL, GrB_PLUS_INT8, GrB_PLUS_UINT8, GrB_PLUS_INT16, GrB_PLUS_UINT16, GrB_PLUS_INT32, GrB_PLUS_UINT32, GrB_PLUS_INT64, GrB_PLUS_UINT64, GrB_PLUS_FP32, GrB_PLUS_FP64.

- There are **over 1000** combinations of predefined operators and types.

- Creates a large burden on implementers, who mostly resort to automatic code generation.
C API: Quality of Life Issues

- For each predefined **GraphBLAS operator**, the C API requires a separate C function for each of 11 predefined types:

GrB_PLUS_BOOL, GrB_PLUS_INT8, GrB_PLUS_UINT8, GrB_PLUS_INT16, GrB_PLUS_UINT16, GrB_PLUS_INT32, GrB_PLUS_UINT32, GrB_PLUS_INT64, GrB_PLUS_UINT64, GrB_PLUS_FP32, GrB_PLUS_FP64.

- There are **over 1000** combinations of predefined operators and types.

- Creates a large burden on implementers, who mostly resort to automatic code generation.
C API: Quality of Life Issues

- For each predefined GraphBLAS operator, the C API requires a separate C function for each of 11 predefined types:
  - GrB_PLUS_BOOL, GrB_PLUS_INT8, GrB_PLUS_UINT8, GrB_PLUS_INT16, GrB_PLUS_UINT16, GrB_PLUS_INT32, GrB_PLUS_UINT32, GrB_PLUS_INT64, GrB_PLUS_UINT64, GrB_PLUS_FP32, GrB_PLUS_FP64.

- There are over 1000 combinations of predefined operators and types.

- Creates a large burden on implementers, who mostly resort to automatic code generation.

Descriptors

Flags for transformations

Transpose, complement, structure-only, etc.
C API: Quality of Life Issues

- User-defined types **must be trivially copyable types** (i.e. memcpy-able).
  ```c
  struct MyComplex {
    int ireal; int iimag;
  };
  ```
  This simplifies API and improves performance, but **limits expressiveness**.
  ```c
  GrB_Type complex_type;
  GrB_Type_new(&complex_type, sizeof(MyComplex));
  GrB_Matrix A;
  GrB_Matrix_new(&A, complex_type, 100, 100);
  ```
- Users have already run into cases where they wish to use **more complex types**.
C API: Quality of Life Issues

- User-defined types **must be trivially copyable types** (i.e. memcpy-able).
  
  ```c
  struct MyComplex {
    int ireal; int iimag;
  };
  ```

- This simplifies API and improves performance, but **limits expressiveness**.
  
  ```c
  GrB_Type complex_type;
  GrB_Type_new(&complex_type,
        sizeof(MyComplex));
  GrB_Matrix A;
  GrB_Matrix_new(&A, complex_type, 100, 100);
  ```

- Users have already run into cases where they wish to use **more complex types**.
C API: Quality of Life Issues

● User-defined types **must be trivially copyable types** (i.e. memcpy-able).

```c
struct MyComplex {
    int ireal; int iimag;
};
```

● This simplifies API and improves performance, but **limits expressiveness**.

```c
GrB_Type complex_type;
GrB_Type_new(&complex_type,
             sizeof(MyComplex));
GrB_Matrix A;
GrB_Matrix_new(&A, complex_type, 100, 100);
```

● Users have already run into cases where they wish to use **more complex types**.
C API: Issues with Types

C API users pass **function pointers** to custom operators

```c
void scale_2(void *out, const void *in) {
    *(int*)out = 2 * (*(int*)in);
}
```

```c
GrB_UnaryOp my_scale_2;
GrB_UnaryOp_new(&my_scale_2, scale_2,
                GrB_INT32, GrB_INT32);
```

**Required for any operator** on user-defined types, but also allows for **operators on built-in types** left out of the spec

Function pointers (e.g. `scale_2`) then used in **performance-critical** inner loops:

```c
GrB_apply(C, ..., my_scale_2, A, desc);
```
C API: Issues with Types

C API users pass **function pointers** to custom operators

```c
void scale_2(void *out, const void *in) {
    *(int*)out = 2 * (*(int*)in);
}

GrB_UnaryOp my_scale_2;
GrB_UnaryOp_new(&my_scale_2, scale_2,
                  GrB_INT32, GrB_INT32);
```

**Required for any operator** on user-defined types, but also allows for **operators on built-in types** left out of the spec

Function pointers (e.g. scale_2) then used in **performance-critical** inner loops:

```c
GrB_apply(C, ..., my_scale_2, A, desc);
```
Drafting a GraphBLAS C++ API
C++ Has a Rich Type System

- User-defined types are **first-class types**

- They simply need be **copy constructible**, etc.

- Things like **views** can simplify APIs
C++ Has a Rich Type System

- User-defined types are **first-class types**

- They simply need be **copy constructible**, etc.

- Things like **views** can simplify APIs
C++ Has a Rich Type System

- User-defined types are **first-class types**

- They simply need be **copy constructible, etc.**

- Things like **views** can simplify APIs
Disclaimer: API in Progress

- The GraphBLAS C++ API is still in **draft** process

- Specific **names** and **APIs** may change

- There are currently two **draft implementations**, **GBTL** and **RGRI**

- Some slide contents **may be in RGRI**, but not necessarily in C++ spec (yet)
GraphBLAS Concepts

Algorithms
- Generalized Matrix Multiply
- Elementwise Ops

Monoid
- +
- \(i: 0\)

Binary Op
- \(/

Semiring
- *
- +
- \(i: 0\)

Matrix

Transpose View

Mask

Vector

GraphBLAS Concepts
GraphBLAS Concepts

Algorithms

- Generalized Matrix Multiply
- Elementwise Ops

Monoid

- Binary Op
  - +
  - /

Semiring

- * → +
- i: 0

GraphBLAS Concepts

Transpose View

- Matrix

- Vector

Semiring

- * → +
- i: 0
GraphBLAS Concepts

Algorithms
- Generalized Matrix Multiply
- Elementwise Ops

Monoid
- +
- / $i:0$

Binary Op
- $*$
- $+$ $i:0$

Semiring

Matrix
- Transpose View
- Mask
- Vector

*GraphBLAS Concepts*
GraphBLAS Concepts

**Algorithms**
- Generalized Matrix Multiply
- Elementwise Ops

**Monoid**
- Binary Op
- $i: 0$

**Semiring**
- $i: 0$

**Matrix**
- **Transpose View**
- **Mask**
- **Vector**

Carnegie Mellon University
Software Engineering Institute

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.
GraphBLAS Concepts

Algorithms

- Generalized Matrix Multiply
- Elementwise Ops

Monoid

- Binary Op

Semiring

- *  
- +  

Matrix

- Transpose View
- Mask
- Vector
GraphBLAS Matrix

- A matrix is a collection of **stored values**
- It has a **shape** (number of rows, cols)
- It has a **size** (number of stored values)
- Can **access individual locations**
- Can **iterate** over values
GraphBLAS Matrix

- A matrix is a collection of **stored values**
- It has a **shape** (number of rows, cols)
- It has a **size** (number of stored values)
- Can **access** individual locations
- Can **iterate** over values
GraphBLAS Matrix

- A matrix is a collection of *stored values*
- It has a **shape** (number of rows, cols)
- It has a **size** (number of stored values)
- Can **access** individual locations
- Can **iterate** over values
GraphBLAS Matrix

- A matrix is a collection of **stored values**
- It has a **shape** (number of rows, cols)
- It has a **size** (number of stored values)
- Can **access individual locations**
- Can **iterate** over values
GraphBLAS Matrix

- A matrix is a collection of **stored values**
- It has a **shape** (number of rows, cols)
- It has a **size** (number of stored values)
- Can **access individual locations**
- Can **iterate** over values
GraphBLAS Matrix

- A matrix is a collection of **stored values**
- It has a **shape** (number of rows, cols)
- It has a **size** (number of stored values)
- Can **access** individual locations
- Can **iterate** over values

Not included: implicit zero value!
Sparse Matrix - Similarities to `std::unordered_map`

- Distinct set of keys

- Each key associated with a value

- Individual lookup/insertion by key

- Iteration over unordered range of values

```
[“x”] 120
[“z”] 122
[“y”] 121
[“b”] 98
```
Sparse Matrix - Similarities to \texttt{std::unordered_map}

- Distinct \textbf{set of keys}

- Each key \textbf{associated} with a value

- Individual \textbf{lookup/insertion} by key

- \textbf{Iteration} over unordered range of values

\begin{verbatim}
[x] 120
[z] 122
[y] 121
[b] 98
\end{verbatim}
Sparse Matrix - Similarities to `std::unordered_map`

- Distinct set of keys

- Each key associated with a value

- Individual lookup/insertion by key

- Iteration over unordered range of values

```
[“x”] → 120
[“z”] → 122
[“y”] → 121
[“b”] → 98
```
Sparse Matrix - Similarities to `std::unordered_map`

- Distinct *set of keys*

- Each key *associated* with a value

- Individual *lookup*/*insertion* by key

- *Iteration* over unordered range of values

```
["x"]    120
["z"]    122
["y"]    121
["b"]    98
```
Sparse Matrix - *Differences* from std::unordered_map

- **key_type** is pair-like type filled with integral values

- Matrix shape restricts valid key values

- Implementation will use highly specialized sparse matrix formats

- Indices and value may not be materialized in memory

```c++
{0, 1} 120
{2, 3} 122
{4, 3} 121
{7, 0} 98
```
Sparse Matrix - *Differences* from `std::unordered_map`

- **key_type** is **pair-like type** filled with integral values
  - `{{0, 1}}` → 120

- **Matrix shape** restricts **valid key values**
  - `{{2, 3}}` → 122
  - `{{4, 3}}` → 121
  - `{{7, 0}}` → 98

- **Implementation** will use highly specialized sparse matrix formats

- **Indices** and **value** may not be materialized in memory
Sparse Matrix - *Differences* from std::unordered_map

- **key_type** is pair-like type filled with
  integral values

  $\{0, 1\}$

  120

- **Matrix shape** restricts valid key values

  $\{2, 3\}$

  122

- **Implementation** will use highly
  specialized sparse matrix formats

  $\{4, 3\}$

  121

  $\{7, 0\}$

  98

- **Indices** and **value** may not be
  materialized in memory
Sparse Matrix - *Differences* from std::unordered_map

- `key_type` is pair-like type filled with integral values
  - \([\{0, 1\}] \rightarrow 120\)

- Matrix shape restricts valid key values
  - \([\{2, 3\}] \rightarrow 122\)

- Implementation will use highly specialized sparse matrix formats
  - \([\{4, 3\}] \rightarrow 121\)
  - \([\{7, 0\}] \rightarrow 98\)

- Indices and value may not be materialized in memory
Sparse Matrix - *Differences* from `std::unordered_map`

```cpp
using key_type = std::pair<int, int>;
using map_type = int;

unordered_map<key_type, map_type> x = ...;

auto iter = x.begin();

[blank] value = *iter;
```

```plaintext
{{0, 1}] 120
{{2, 3}] 122
{{4, 3}] 121
{{7, 0}] 98
```
Sparse Matrix - *Differences* from `std::unordered_map`

```cpp
using key_type = std::pair<int, int>;
using map_type = int;

unordered_map<key_type, map_type> x = ...;
auto iter = x.begin();
[blank] value = *iter;
```

What is the type of `*iter`?

- `{0, 1}`: 120
- `{2, 3}`: 122
- `{4, 3}`: 121
- `{7, 0}`: 98
Sparse Matrix - *Differences* from `std::unordered_map`

```cpp
using key_type = std::pair<int, int>;
using map_type = int;

unordered_map<key_type, map_type> x = ...;

auto iter = x.begin();
[blank] value = *iter;
```

```plaintext
[{0, 1}] 120
[{2, 3}] 122
[{4, 3}] 121
[{7, 0}] 98
```
Sparse Matrix - *Differences* from `std::unordered_map`

```cpp
using key_type = std::pair<int, int>
using map_type = int;

unordered_map<key_type, map_type> x = ...;

auto iter = x.begin();

using value_type = std::pair<const key_type, map_type>
value_type& value = *iter;
```

```plaintext
[0, 1]  120
[2, 3]  122
[4, 3]  121
[7, 0]  98
```
Sparse Matrix - *Differences* from std::unordered_map

```cpp
using key_type = std::pair<int, int>;  
using map_type = int;

unordered_map<key_type, map_type> x = ...;

auto iter = x.begin();

using value_type = std::pair<const key_type, map_type>;
value_type& value = *iter;
```

- `{0, 1}` → 120
- `{2, 3}` → 122
- `{4, 3}` → 121
- `{7, 0}` → 98
Sparse Matrix - *Differences* from `std::unordered_map`

1. Each element exists materialized somewhere
2. Can obtain int& reference to value, const pair<...>& reference to key.

```
using key_type = int;
using map_type = int;
unordered_map<key_type, map_type> x = ...;
auto iter = x.begin();
using value_type = std::pair<const key_type, map_type>;
value_type& value = *iter;
```

(Possible) Physical Memory Layout

```
{{0, 1}, 120}  {{4, 3}, 121}  {{7, 0}, 98}  {{2, 3}, 122}
```
Sparse Matrix - *Differences* from std::unordered_map

1. Each element exists **materialized** somewhere

2. Can obtain `int&` reference to value, `const pair<...>&` reference to key.

```cpp
using key_type = std::pair<int, int>;
using map_type = int;
unordered_map<key_type, map_type> x = ...

auto iter = x.begin();

using value_type = std::pair<const key_type, map_type>;
value_type& value = *iter;
```

**Physical Memory Layout**

| [0, 1]  | 120 |
| [2, 3]  | 122 |
| [4, 3]  | 121 |
| [7, 0]  | 98  |
| [2, 3]  | 122 |
Sparse Matrix - *Differences* from std::unordered_map

1. Each element exists *materialized* somewhere
2. Can obtain `int&` reference to value, `const pair<...>&` reference to key.

```cpp
using key_type = std::pair<int, int>;
using map_type = int;
unordered_map<key_type, map_type> x;
auto iter = x.begin();
using value_type = std::pair<const key_type, map_type>;
value_type& value = *iter;
```

(Possible) Physical Memory Layout

| {0, 1} | 120 |
| {2, 3} | 122 |
| {4, 3} | 121 |
| {7, 0} | 98  |
| {2, 3} | 122 |
Sparse Matrix - *Differences* from `std::unordered_map`

1. Each element exists materialized somewhere

2. Can obtain `int&` reference to value, `const pair<...>&` reference to key.

Using:
- `key_type = std::pair<int, int>;
- map_type = int;
- unordered_map<key_type, map_type> x = ...;
- auto iter = x.begin();
- using value_type = std::pair<const key_type, map_type>;
- value_type& value = *iter;

(Possible) Physical Memory Layout:

- `{0, 1}` → 120
- `{2, 3}` → 122
- `{4, 3}` → 121
- `{7, 0}` → 98

```cpp
const pair<int, int>&
```
Sparse Matrix - *Differences* from `std::unordered_map`

1. Each element exists materialized somewhere

2. Can obtain `int&` reference to value, `const pair<...>&` reference to key

```cpp
using key_type = std::pair<int, int>;
using map_type = int;

unordered_map<key_type, map_type> x = ...;

auto iter = x.begin();

using value_type = std::pair<const key_type, map_type>;

value_type& value = *iter;
```

(Possible) Physical Memory Layout

<table>
<thead>
<tr>
<th>{[0, 1]}</th>
<th>{[2, 3]}</th>
<th>{[4, 3]}</th>
<th>{[7, 0]}</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>122</td>
<td>121</td>
<td>98</td>
</tr>
</tbody>
</table>

```cpp
int&
```
Sparse Matrix Formats

- Need to enable a variety of different sparse matrix formats

- Most formats separate values and indices, may not store some indices

- This means we need to use a custom reference type for indices
Sparse Matrix Formats

- Need to enable a **variety** of different sparse matrix formats

- Most formats separate **values** and **indices**, may not store some indices

- This means we need to use a **custom** reference type for indices

### Compressed Sparse Row (CSR) Storage Format

<table>
<thead>
<tr>
<th>Row Pointers</th>
<th>Column Indices</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3 3 3 6 6 7</td>
<td>0 2 2 2 3 4 3</td>
<td>8 2 5 7 1 2 9</td>
</tr>
</tbody>
</table>

Sparse Matrix Representation

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Sparse Matrix Formats

- Need to enable a *variety* of different sparse matrix formats

- Most formats separate values and indices, may not store some indices

- This means we need to use a *custom* reference type for indices
Sparse Matrix Formats

- Need to enable **a variety** of different sparse matrix formats

- Most formats **separate values** and **indices**, may not store some indices

- This means we need to use a **custom reference type** for indices

Compressed Sparse Row (CSR) Storage Format

<table>
<thead>
<tr>
<th>Row Pointers</th>
<th>Column Indices</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3 3 6 6 7</td>
<td>0 2 2 3 4 3</td>
<td>8 2 5 7 1 2 9</td>
</tr>
</tbody>
</table>

Sparse Matrix Representation

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>7 1 2</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Sparse Matrix Formats

- Need to enable a variety of different sparse matrix formats

- Most formats separate values and indices, may not store some indices

- This means we need to use a custom reference type for indices

Compressed Sparse Row (CSR) Storage Format

Row Pointers

| 0 | 2 | 3 | 3 | 6 | 6 | 7 |

Column Indices

| 0 | 2 | 2 | 2 | 3 | 4 | 3 |

Values

| 8 | 2 | 5 | 7 | 1 | 2 | 9 |

Sparse Matrix Representation

| 8 | 2 |
|----|
| 5 | 7 |
| 1 | 1 | 2 |
Sparse Matrix Formats

- Need to enable a **variety** of different sparse matrix formats

- Most formats **separate** values and **indices**, may

- This means we need to use a **custom** reference type for indices

Compressed Sparse Row (CSR) Storage Format

<table>
<thead>
<tr>
<th>Row Pointers</th>
<th>Column Indices</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3 3 3 6 6 7</td>
<td>0 2 2 2 3 4 3</td>
<td>8 2 5 7 1 2 9</td>
</tr>
</tbody>
</table>

Custom reference type, like `vector<bool>::reference`
Matrix Data Structure

grb::matrix<float>

Type of stored values
Matrix Data Structure

grb::matrix<float, int>

Type of stored values (Integer) type used to store indices
Matrix Data Structure: Attributes

**Attributes**

**Shape**
- **Dimensions of matrix**
  (Graph: number of vertices)

**Size**
- **Number of stored values**
  (Graph: number of edges)

```cpp
grb::matrix<float> x({1024, 1024});
size_t m = x.shape()[0];
size_t n = x.shape()[1];
size_t nnz = x.size();
```
Matrix Data Structure: Element Access

**Element Access**
Direct access to stored values

**operator[]**
Find or insert value by index

**find**
Find value by index

```cpp
grb::matrix<float, int> m({1024, 1024});

m[{0, 0}] = 12;
m[{1, 1}] = 12;
m[{2, 2}] = 12;
m[{3, 3}] = 12;

if (m.find({3, 3}) != m.end()) {
  // Should run, just set elem 3, 3 to 12.
}

if (m.find({4, 4}) != m.end()) {
  // Will not run, have not yet set elem 4, 4
}
```
Matrix Data Structure: Iteration

**Iteration**
Iteration over stored values

**Can read:** row, column, value

**Can write:** value only

Iteration allows support for standard C++ algorithms.

```cpp
grb::matrix<float, int> m = ...;

for (auto iter = m.begin(); iter != m.end(); ++iter) {
    float x = *iter;
}

for (auto&& [i, j, v] : m) {
    v = 12;
    printf("Elem. %d, %d set to %f\n", i, j, v);
}

std::reduce(m.begin(), m.end(), float(0));
```
Sparse Matrix Formats

- Many potential *sparse* matrix formats

- Each format has different iteration patterns

- Inefficient to enforce a particular iteration order

Sparse Matrix Formats

- Many potential sparse matrix formats

- Each format has different iteration patterns

- Inefficient to enforce a particular iteration order
Sparse Matrix Formats

- Many potential sparse matrix formats
- Each format has different iteration patterns
- Inefficient to enforce a particular iteration order

Matrix Data Structure

grb::matrix<float, int, grb::column>

- Type of stored values
- (Integer) type used to store indices
- Compile-time hint about storage format
Matrix Data Structure: Iteration

- **Unordered iteration** over stored values

- Range of `size()` `matrix_entry<T, I>` elements

- **Tuple-like type** with access to indices and T& reference to value

```
grb::matrix<float, int> m = ...;

for (auto iter = m.begin(); iter != m.end(); ++iter) {
    float x = *iter;
}

for (auto&& [i, j, v] : m) {
    v = 12;
    printf("Elem. %d, %d set to %f\n", i, j, v);
}

std::reduce(m.begin(), m.end(), float(0));
```
Matrix Data Structure: Iteration

- **Unordered iteration** over stored values

- Range of size() `matrix_entry<T, I>` elements

- Tuple-like type with access to indices and T& reference to value

```cpp
grb::matrix<float, int> m = ...;

for (auto iter = m.begin(); iter != m.end(); ++iter) {
    float x = *iter;
}

for (auto&& [i, j, v] : m) {
    v = 12;
    printf("Elem. %d, %d set to %f\n", i, j, v);
}

std::reduce(m.begin(), m.end(), float(0));
```
Matrix Data Structure: Iteration

- **Unordered iteration** over stored values

- Range of `size()` `matrix_entry<T, I>` elements

- **Tuple-like type** with access to indices and `T&` reference to value

```
grb::matrix<float, int> m = ...;

for (auto iter = m.begin(); iter != m.end(); ++iter) {
    float x = *iter;
}
```

```
for (auto&& [i, j, v] : m) {
    v = 12;
    printf("Elem. %d, %d set to %f\n", i, j, v);
}
```

```
std::reduce(m.begin(), m.end(), float(0));
```
GraphBLAS Concepts

Algorithms

- Generalized Matrix Multiply
- Elementwise Ops

Monoid

- Binary Op

Semiring

- +
- /

GraphBLAS Concepts

Matrix

- Transpose View
- Mask
- Vector
Binary Operators

**Functors** that operate on **two** inputs, producing a single output

\[ T \times U \rightarrow V \]

**Rule:** types \( T, U, \) and \( V \) are determined by matrices. Op. must accept \( T, U, V \).

```cpp
grb::ewise_add(c, ..., a, b,
               std::plus<int>());

auto my_op = [](auto a, auto b) {
    return a*b + 2;
};

grb::ewise_mult(c, ..., a, b, my_op);
```
Monoids: Binary Operators with an Identity

- Monoids are **mathematical objects**, consisting of:
  
  - A **commutative** binary operator
  
  - A type \( T \)
  
  - A mathematical **identity**
Monoids: Binary Operators with an Identity

- Monoids are **mathematical objects**, consisting of:
  - A **commutative** binary operator
  - A type $T$
  - A mathematical **identity**
Monoids: Binary Operators with an Identity

- Monoids are mathematical objects, consisting of:

- A commutative binary operator

- A type T

- A mathematical identity
Monoids: Binary Operators with an Identity

- Monoids are **mathematical objects**, consisting of:

  - A **commutative** binary operator
  - A type $T$
  - A mathematical **identity**
Monoids

- Given a **binary operator fn** and a **type T**, we can ask:

  Does binary op. fn form a monoid on type T?

- Depends on whether **monoid_traits** specialization exists

```cpp
using grb;

bool test = is_monoid_v<std::plus<>, int>;

// Prints "1" for true
std::cout << test << std::endl;

int identity = monoid_traits<std::plus<>,
                             int>::identity();

// Prints "0", since identity for std::plus<> on type `int` is `0`
std::cout << identity << std::endl;
```
Monoids

- Given a **binary operator fn** and a **type T**, we can ask:

  Does binary op. fn form a monoid on type T?

- Depends on whether **monoid_traits** specialization exists

```cpp
using grb;

bool test = is_monoid_v<std::plus<int>, int>;

// Prints "1" for true
std::cout << test << std::endl;

int identity = monoid_traits<std::plus<int>, int>::identity();

// Prints "0", since identity for std::plus<int> on type `int` is `0`
std::cout << identity << std::endl;
```
Monoids

- Given a binary operator fn and a type T, we can ask:
  
  Does binary op. fn form a monoid on type T?
  
- Depends on whether monoid_traits specialization exists

```cpp
using grb;

bool test = is_monoid_v<std::plus<int>>;

// Prints “1” for true
std::cout << test << std::endl;

int identity = monoid_traits<std::plus<int>>::identity();

// Prints “0”, since identity for std::plus<int> on type `int` is `0`
std::cout << identity << std::endl;
```
Monoids

- Given a **binary operator** `fn` and a type `T`, we can ask:
  Does binary op. `fn` form a monoid on type `T`?

- Depends on whether **monoid_traits** specialization exists

```cpp
using grb;

bool test = is_monoid_v< std::plus<> , int > ;

// Prints "1" for true
std::cout << test << std::endl;

int identity = monoid_traits< std::plus<> , int >::identity();

// Prints "0", since identity for std::plus<> on type `int` is `0`
std::cout << identity << std::endl;
```
Obtaining a Monoid

- Use **pre-defined binary ops** such as `grb::plus`, `grb::multiplies`

- Define a **specialization** of `grb::monoid_traits`

- Add **identity()** method to op

- Use **make_monoid** helper function

```cpp
using grb;

// Using a pre-defined binary op
grb::plus<> fn;
std::plus<> fn_stl;

bool g = is_monoid<grb::plus<> , int>::value;
bool s = is_monoid<std::plus<> , int>::value;

std::cout << g << " " << s << std::endl;
```
Obtaining a Monoid

- Use **pre-defined binary ops** such as `grb::plus`, `grb::multiplies`

- Define a **specialization** of `grb::monoid_traits`

- Add `identity()` method to op

- Use `make_monoid` helper function

```cpp
using grb;

// Using a pre-defined binary op
grb::plus<> fn;
std::plus<> fn_stl;

bool g = is_monoid<grb::plus<>, int>::value;
bool s = is_monoid<std::plus<>, int>::value;

std::cout << g << " " << s << std::endl;
```
Obtaining a Monoid

- Use **pre-defined binary ops** such as grb::plus, grb::multiplies

- Define a **specialization** of grb::monoid_traits

- Add **identity() method** to op

- Use **make_monoid** helper function

```cpp
struct my_plus {
    float operator()(float a, float b) {
        return a + b;
    }

    float identity() {
        return 0.0f;
    }
};

... i =
    grb::monoid_traits<my_plus, int>::identity();
```
Obtaining a Monoid

- Use **pre-defined binary ops** such as `grb::plus`, `grb::multiplies`

- Define a **specialization** of `grb::monoid_traits`

- Add **identity() method** to `op`

- Use **`make_monoid`** helper function

```cpp
struct my_plus {
  float operator()(float a, float b) {
    return a + b;
  }

  float identity() {
    return 0.0f;
  }
};

int i = grb::monoid_traits<my_plus, int>::identity();
```
Obtaining a Monoid

- Use **pre-defined binary ops** such as `grb::plus`, `grb::multiplies`
- Define a **specialization** of `grb::monoid_traits`
- Add **identity()** method to `op`
- Use **`make_monoid`** helper function

```cpp
auto my_op = [](auto a, auto b) {
    return a * b;
};

auto my_monoid = make_monoid(my_op, 1);
```
Semirings

Semirings combine a binary op b and a monoid m, where b distributes over m

1) **Pre-define** a number of semirings

2) Users can **build semirings** with make_semiring

```cpp
auto semiring =
    grb::plus_multiplies_semiring();

auto my_times = [] (auto a, auto b) {
    return a*b;
};

auto my_plus = [] (auto a, auto b) {
    return a+b;
};

auto m_plus = grb::make_monoid(my_plus, 0);

auto my_semiring =
    grb::make_semiring(m_plus, my_times);
```
Semirings

Semirings combine a **binary op** \( b \) and a **monoid** \( m \), where \( b \) distributes over \( m \)

1) **Pre-define** a number of semirings

2) Users can **build semirings** with `make_semiring`

```
auto semiring =
  grb::plus_multiplies_semiring();

auto my_times = [](auto a, auto b) {
  return a*b;
};

auto my_plus = [](auto a, auto b) {
  return a+b;
};

auto m_plus = grb::make_monoid(my_plus, 0);

auto my_semiring =
  grb::make_semiring(m_plus, my_times);
```
Semirings

Semirings combine a binary op $b$ and a monoid $m$, where $b$ distributes over $m$

1) **Pre-define** a number of semirings

2) Users can **build semirings** with `make_semiring`

```cpp
auto semiring =
    grb::plus_multiplies_semiring();

auto my_times = [](auto a, auto b) {
    return a*b;
};

auto my_plus = [](auto a, auto b) {
    return a+b;
};

auto m_plus = grb::make_monoid(my_plus, 0);

auto my_semiring =
    grb::make_semiring(m_plus, my_times);
```
Semirings

Semirings combine a **binary op b** and a **monoid m**, where **b** distributes over **m**

1) **Pre-define** a number of semirings

2) Users can **build semirings** with **make_semiring**

```cpp
auto semiring =
    grb::plus_multiplies_semiring();

auto my_times = [] (auto a, auto b) {
    return a*b;
};

auto my_plus = [] (auto a, auto b) {
    return a+b;
};

auto m_plus = grb::make_monoid(my_plus, 0);

auto my_semiring =
    grb::make_semiring(m_plus, my_times);
```
Semirings combine a binary op \( b \) and a monoid \( m \), where \( b \) distributes over \( m \)

1) **Pre-define** a number of semirings

2) Users can **build semirings** with `make_semiring`

```cpp
auto semiring =
    grb::plus_multiplies_semiring();

auto my_times = [](auto a, auto b) {
    return a*b;
};

auto my_plus = [](auto a, auto b) {
    return a+b;
};

auto m_plus = grb::make_monoid(my_plus, 0);

auto my_semiring =
    grb::make_semiring(m_plus, my_times);
```
Semirings

Semirings combine a binary op b and a monoid m, where b distributes over m

1) **Pre-define** a number of semirings

2) Users can **build semirings** with `make_semiring`

```cpp
auto semiring =
    grb::plus_multiplies_semiring();

auto my_times = [](auto a, auto b) {
    return a*b;
};

auto my_plus = [](auto a, auto b) {
    return a+b;
};

auto m_plus = grb::make_monoid(my_plus, 0);

auto my_semiring =
    grb::make_semiring(m_plus, my_times);
```
Semirings

Semirings combine a binary op \( b \) and a monoid \( m \), where \( b \) distributes over \( m \).

1) **Pre-define** a number of semirings

2) Users can **build semirings** with `make_semiring`

```cpp
auto semiring =
    grb::plus_multiplies_semiring();

auto my_times = [](auto a, auto b) {
    return a*b;
};

auto my_plus = [](auto a, auto b) {
    return a+b;
};

auto m_plus = grb::make_monoid(my_plus, 0);

auto my_semiring =
    grb::make_semiring(m_plus, my_times);
```
GraphBLAS Concepts

Algorithms
- Generalized Matrix Multiply
- Elementwise Ops

Monoid
- Binary Op
- Semiring

Matrix

Transpose View

Mask

Vector
Views

Views provide a (typically transformed) view of a matrix.

We can create views representing transpose, structure, complement, etc.

This simplifies API, removes some of need for descriptors.

grb::matrix<float> a = ...;

auto a_t = grb::transpose(a);

auto b = grb::multiply(a, a_t);
Views

Views provide a (typically transformed) view of a matrix.

We can create views representing transpose, structure, complement, etc.

This simplifies API, removes some of need for descriptors.

\[
\begin{align*}
\text{grb::matrix<float> } a &= \ldots; \\
\text{auto } a_t &= \text{grb::transpose}(a); \\
\text{auto } b &= \text{grb::multiply}(a, a_t);
\end{align*}
\]
Matrix Transform Views

Provide a **const view of a matrix** with each stored value transformed

- Can be used to create structure-only view

```cpp
grb::matrix<float> a = ...;

auto t = [](grb::matrix_entry<float> e) {
    return true;
};

auto a_t = grb::transform_view(a, t);

for (auto&& [i, j, v] : a_t) {
    printf("Elem (%d, %d): %f\n", i, j, v);
}
```
Matrix Transform Views

Provide a **const view of a matrix** with each stored value transformed

- Can be used to create structure-only view

```cpp
grb::matrix<float> a = ...;

auto t = 
    [](grb::matrix_entry<float> e) {
        return true;
    };

auto a_t = grb::transform_view(a, t);

for (auto&& [i, j, v] : a_t) {
    printf("Elem (%d, %d): %f\n", i, j, v);
}
```
Matrix Transform Views

Provide a **const view of a matrix** with each stored value **transformed**

- Can be used to create **structure-only view**

```cpp
Matrix<float> a = ...;

auto t = grb::matrix_entry<float>(e) {
    return true;
};

grb::transform_view(a, t);

for (auto&& [i, j, v] : a_t) {
    printf("Elem (%d, %d): %f\n", i, j, v);
}
```

Matrix A

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Matrix Transform Views

Provide a **const view of a matrix** with each stored value transformed

- Can be used to create **structure-only view**

```cpp
MatrixTransformViews

Matrix A

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

view

true  true  true
true  true
true

int main() {
    matrix<float> a = ...
    auto t = [] (matrix_entry<float> e) {
        return true;
    };
    auto a_t = grb::transform_view(a, t);
    for (auto&& [i, j, v] : a_t) {
        printf("Elem (%d, %d): %f\n", i, j, v);
    }
    return 0;
}
```
GraphBLAS Masks

- **Range** of matrix elements

- **Element-wise** access methods

- **Shape**

- Stored values *convertible to* *bool*
GraphBLAS Masks

- **Range** of matrix elements
- **Element-wise** access methods
- **Shape**
- Stored values **convertible to bool**
GraphBLAS Masks

- **Range** of matrix elements
- **Element-wise** access methods
- **Shape**
- Stored values **convertible to bool**
GraphBLAS Masks

- **Range** of matrix elements
- **Element-wise** access methods
- **Shape**
- Stored values **convertible to bool**
GraphBLAS Masks

- **Range** of matrix elements
- **Element-wise** access methods
- **Shape**
- Stored values **convertible to bool**
GraphBLAS Masks

- **Range** of matrix elements
- **Element-wise** access methods
- **Shape**
- Stored values *convertible to bool*

![Diagram](image-url)
GraphBLAS Masks

- **Range** of matrix elements

- **Element-wise** access methods

- **Shape**

- Stored values **convertible to bool**
GraphBLAS Masks

- **Range** of matrix elements
- **Element-wise** access methods
- **Shape**

- Stored values *convertible to bool*
GraphBLAS Masks

- **Range** of matrix elements
- **Element-wise** access methods
- **Shape**

- Stored values convertible to bool

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Apply output mask

\[
x \times m = \\
\begin{array}{c}
\begin{array}{c}
1 \\
1 \\
1 \\
1 \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
1 \\
1 \\
0 \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
10 \\
5 \\
\end{array}
\end{array}
\end{array}
\]
GraphBLAS Masks

- **Range** of matrix elements
- **Element-wise** access methods
- **Shape**
- Stored values **convertible to bool**

Apply output mask
GraphBLAS Concepts

Algorithms
- Generalized Matrix Multiply
- Elementwise Ops

Monoid
- Binary Op:
  - Addition (+)
  - Division (/)

Semiring
- Multiplication (*)
- Addition (+)

Matrix
- Transpose View
- Mask
- Vector

GraphBLAS Concepts

Carnegie Mellon University
Software Engineering Institute
Algorithms

The primary algorithms of interest are:

1) **Generalized matrix multiplication** -- using **mask** and **arbitrary semiring**

2) **Elementwise** operations
Matrix Multiply

Accepts matrices, mask, semiring, accumulator, and flag to control merge behavior

Input matrices could be grb::matrix or views

Similar to C API

```cpp
using grb;

matrix<float> a = get_matrix(...);
matrix<float> b = get_matrix(...);

matrix<float> c({a.shape()[0], b.shape()[1]});

mxm(c, plus<>{}, a, b,
    no_mask{}, plus_multiplies_semiring{});
```
Matrix Multiply

Accepts **matrices**, **mask**, **semiring**, **accumulator**, and **flag** to control merge behavior

Input matrices could be `grb::matrix` or views

Similar to C API

```cpp
using grb;

matrix<float> a = get_matrix(...);
matrix<float> b = get_matrix(...);

matrix<float> c({a.shape()[0], b.shape()[1]});

mxm(c, plus>{{}, a, b,
    no_mask{{}, plus_multiplies_semiring{});
```
Matrix Multiply

Accepts **matrices**, **mask**, **semiring**, **accumulator**, and **flag** to control merge behavior.

Input matrices could be **grb::matrix** or **views**.

Similar to C API:

```cpp
using grb;

matrix<float> a = get_matrix(...);
matrix<float> b = get_matrix(...);

matrix<float> c({a.shape()[0], b.shape()[1]});

mxm(c, plus<>{}, a, b,
    no_mask{}, plus_multiplies_semiring{});
```
Matrix Multiply

Accepts matrices, mask, semiring, accumulator, and flag to control merge behavior

Input matrices could be grb::matrix or views

Similar to C API

```cpp
using grb;

matrix<float> a = get_matrix(...);
matrix<float> b = get_matrix(...);

matrix<float> c({a.shape()[0], b.shape()[1]});

mxm(c, plus>{{}, a, b,
        no_mask{{}}, plus_multiples_semiring{}});
```
Matrix Multiply

Accepts matrices, mask, semiring, accumulator, and flag to control merge behavior

Input matrices could be grb::matrix or views

Similar to C API

```cpp
using grb;

matrix<float> a = get_matrix(...);
matrix<float> b = get_matrix(...);

matrix<float> c({a.shape()[0], b.shape()[1]});

mxm(c, plus<>{}, a, b,
    no_mask{}, plus_multiples_semiring{});
```
Matrix Multiply

Accepts matrices, mask, semiring, accumulator, and flag to control merge behavior.

Input matrices could be grb::matrix or views.

Similar to C API.

```cpp
using grb;

matrix<float> a = get_matrix(...);
matrix<float> b = get_matrix(...);

matrix<float> c({a.shape()[0], b.shape()[1]});

mxm(c, plus<>{}, transpose(a), b,
    no_mask{}, plus_multiplies_semiring{});
```
Matrix Multiply

Accepts **matrices**, **mask**, **semiring**, **accumulator**, and **flag** to control merge behavior

Input matrices could be **grb::matrix** or **views**

Similar to C API

```cpp
using grb;

matrix<float> a = get_matrix(...);
matrix<float> b = get_matrix(...);

matrix<float> c({a.shape()[0], b.shape()[1]});

mxm(c, plus<>{}, transpose(a), b,
    no_mask{}, plus_multiplies_semiring{});
```
## Matrix Multiply Definition

**MatrixRange** - an output range of matrix elements, plus element access and shape

**ConstMatrixRange** - an input range of matrix elements, plus `const` element access and shape

**MaskMatrixRange** - `ConstMatrixRange` with values convertible to `bool`

```cpp
template <typename CMatrixType,
          typename Accumulator,
          typename AMatrixType,
          typename BMATRIX_TYPE,
          typename MaskType,
          typename Semiring>
void mxm(CMatrixType&& c, Accumulator&& acc,
         AMatrixType&& a, BMATRIX_TYPE&& b,
         MaskType&& mask, Semiring&& s);
```
Matrix Multiply Definition

**MatrixRange** - an **output range** of matrix elements, plus **element access** and **shape**

**ConstMatrixRange** - an **input range** of matrix elements, plus **const element access** and **shape**

**MaskMatrixRange** - **ConstMatrixRange** with values **convertible to bool**

```
template<typename CMatrixType,
typename Accumulator,
typename AMatrixType,
typename BMMatrixType,
typename MaskType,
typename Semiring>
void mxm(CMatrixType&& c, Accumulator&& acc,
        AMatrixType&& a, BMMatrixType&& b,
        MaskType&& mask, Semiring&& s);
```
Matrix Multiply Definition

**MatrixRange** - an output range of matrix elements, plus element access and shape

**ConstMatrixRange** - an input range of matrix elements, plus const element access and shape

**MaskMatrixRange** - ConstMatrixRange with values convertible to bool

```cpp
template <MatrixRange C,
    typename Accumulator,
    typename AMatrixType,
    typename BMatrixType,
    typename MaskType,
    typename Semiring>
void mxm(C&& c, Accumulator&& acc,
    AMatrixType&& a, BMatrixType&& b,
    MaskType&& mask, Semiring&& s);
```
Matrix Multiply Definition

**MatrixRange** - an **output range** of matrix elements, plus **element access** and **shape**

**ConstMatrixRange** - an **input range** of matrix elements, plus **const** element access and **shape**

**MaskMatrixRange** - **ConstMatrixRange** with values **convertible to bool**

```cpp
template <MatrixRange C,
typlename Accumulator,
typlename AMatrixType,
typlename BMMatrixType,
typlename MaskType,
typlename Semiring>
void mxm(C&& c, Accumulator&& acc, AMatrixType&& a, BMMatrixType&& b, MaskType&& mask, Semiring&& s);
```
Matrix Multiply Definition

**MatrixRange** - an output range of matrix elements, plus element access and shape

**ConstMatrixRange** - an input range of matrix elements, plus const element access and shape

**MaskMatrixRange** - ConstMatrixRange with values convertible to bool

```cpp
template <MatrixRange C,
          typename Accumulator,
          ConstMatrixRange A,
          ConstMatrixRange B,
          typename MaskType,
          typename Semiring>
void mxm(C&& c, Accumulator&& acc,
         A&& a, A&& b,
         MaskType&& mask, Semiring&& s);
```
Matrix Multiply Definition

**MatrixRange** - an output range of matrix elements, plus **element access** and **shape**

**ConstMatrixRange** - an input range of matrix elements, plus **const element access** and **shape**

**MaskMatrixRange** - ConstMatrixRange with values convertible to bool

---

```cpp
template <MatrixRange C,
    typename Accumulator,
    ConstMatrixRange A,
    ConstMatrixRange B,
    typename MaskType,
    typename Semiring>
void mxm(C&& c, Accumulator&& acc,
    A&& a, A&& b,
    MaskType&& mask, Semiring&& s);
```
Matrix Multiply Definition

MatrixRange - an output range of matrix elements, plus element access and shape

ConstMatrixRange - an input range of matrix elements, plus const element access and shape

MaskMatrixRange - ConstMatrixRange with values convertible to bool

template <MatrixRange C,
    typename Accumulator,
    ConstMatrixRange A,
    ConstMatrixRange B,
    MaskMatrixRange M,
    typename Semiring>

void mxm(C&& c, Accumulator&& acc,
         A&& a, B&& b,
         M&& mask, Semiring&& s);
Matrix Times Matrix

```
grb::matrix<float> c = ...;
grb::matrix<float> a = ...;

auto a_t = grb::transpose(a);
auto mask = grb::structure(c);

grb::mxm(c, mask,
    grb::plus{},
    grb::plus_times_semiring{},
    a, a_t);
```

Matrix Times Matrix (mxm)

Very similar to C API

Accepts matrices, mask, accumulator, semiring, and flag to control merge behavior

Input matrices could be `grb::matrix` or views
Interoperability with C++ Algorithms

- C++ GraphBLAS matrices are ranges, which allows us to use C++ standard algorithms

- Area for exploration: implementing GraphBLAS operations with standard C++ algorithms

- One dimensional iteration somewhat limited, but 2D iteration concepts are coming (next slide)
Interoperability with C++ Algorithms

- C++ GraphBLAS matrices are ranges, which allows us to use C++ standard algorithms

- Area for exploration: implementing GraphBLAS operations with standard C++ algorithms

- One dimensional iteration somewhat limited, but 2D iteration concepts are coming (next slide)
Interoperability with C++ Algorithms

- C++ GraphBLAS matrices are ranges, which allows us to use C++ standard algorithms

- Area for exploration: implementing GraphBLAS operations with standard C++ algorithms

- One dimensional iteration somewhat limited, but 2D iteration concepts are coming (next slide)
Interoperability with C++ Graph Library

- C++ graph library proposal\[P1709\] provides standard concepts for iterating over graphs, graph algorithms

- We aren’t currently using multidimensional iteration

- We should closely examine opportunities for interoperability
  - Implement mxm using graph library concepts
  - Build adapters for graph library concepts to fulfill GrB concepts, vice-versa
Wrap-Up

We can use matrix algebra to implement graph algorithms

Can support a variety of different sparse matrix formats

Provide high-level interfaces for algorithms
Brief Advertisement

If you enjoy parallel programming:

“PGAS in C++: A Portable Abstraction for Distributed Data Structures”

Tuesday, 4:45 PM MDT
Location: D) Valley 1

Virtual: Wednesday, 12:30 PM MDT

GraphBLAS Links

graphblas.org
github.com/cmu-sei/gbtl
github.com/BenBrock/rgri