# GraphBLAS C++ API Specification v1.0

# Intro

The GraphBLAS standard defines a set of generalized matrix and vector operations based on semiring algebraic structures. These operations can be used to express a wide variety of graph algorithms. This specification document defines the C++ programming interface for the GraphBLAS standard, referred to as the . The GraphBLAS C++ API defines a collection of data structures, algorithms, views, operators, and concepts for implementing graph algorithms. These API components are based around mathematical objects, such as matrices and vectors, and operations, such as generalized matrix multiplication and element-wise operations. The GraphBLAS Math Specification provides a complete specification of the mechanics of these structures and operations, and provides a rigorous mathematical definition of the operations and data structures provided in this specification.

# Data Structures

This specification provides data structures for storing matrices and vectors. These implement the matrix and vector range concepts also defined in this specification.

## Matrices

Matrices are two-dimensional collections of sparse values. Each matrix has a defined number of rows and columns, which define its . In addition, each matrix has a defined number of , which are indices inside the bounds of the matrix which actually contain scalar values. A matrix has a fixed , which is the type of the stored values, as well as an , which is the type of the indices.

GraphBLAS matrices are sparse, consist of a well-defined set of stored values, and have no implicit zero value for empty indices. As such, operations may not arbitrarily insert explicit zero values into the matrix, but must produce an output with the exact set of values as specified by the GraphBLAS Math Specification.

The GraphBLAS matrix data structure defined in this specification, `grb::matrix`, provides mechanisms for creating, manipulating, and iterating through GraphBLAS matrices. It also includes a mechanism for influencing the storage format through compile-time hints. GraphBLAS operations that accept matrices can be called with other types so long as they fulfill the GraphBLAS matrix concepts.

## Vectors

Vectors are one-dimensional collections of sparse values. Each vector has a defined number of indices, which defines its shape. Similarly, it has a defined number of stored stored values, which must lie at indices within its shape. Its scalar values have a fixed scalar type, and it has a fixed index type for storing indices. Vectors follow the same sparsity rules as matrices, and similarly the GraphBLAS matrix concepts.

# Concepts

The GraphBLAS C++ Specification defines matrix and vector concepts. These define the interface that a type must support in order to be used by GraphBLAS operations. The intention is that data structures defined by other libraries may be used in GraphBLAS operations so long as they define a small number of customization points that perform insert, find, and iteration operations over the matrix or vector.

## Version Conformance

A C++ GraphBLAS library should define the macro `GRAPHBLAS_CPP_VERSION` to indicate its level of conformance with the GraphBLAS C++ API. To indicate conformance with the GraphBLAS 0.1a draft specification, a library should provide the macro `GRAPHBLAS_CPP_VERSION` with a value greater than or

equal to `202207L`.

**Exposition Only**

```
// Define the `GRAPHBLAS_CPP_VERSION` macro to indicate
// conformance to the GraphBLAS 0.1 Draft Specification.
#define GRAPHBLAS_CPP_VERSION 202207L
```

# Basic Concepts

GraphBLAS defines a collection of objects, functions, and concepts to facilitate the implementation of graph algorithms. These programming facilities are further separated into *GraphBLAS containers*, which include matrices and vectors, *GraphBLAS operations*, which are algorithms such as matrix-vector multiplication that operate over GraphBLAS containers, and *GraphBLAS operators*, which are binary and unary operators that operate over scalar elements that may be held in a container. The C++ API also defines a series of *concepts* defining the interface a container or operator must fulfill in order to to be used with a particular operation, as well as *views*, which provide lazily evaluated, mutated views of a GraphBLAS object.

## GraphBLAS Containers

GraphBLAS containers, which include matrices and vectors, are sparse data structures holding a collection of stored scalar values. Each GraphBLAS container has a shape, which defines the dimension(s) of the object. Stored values can be inserted at indices within the dimensions of the object. Since GraphBLAS containers are sparse, they keep explicit track of which index locations in the object hold stored values. There is no implied zero value associated with a GraphBLAS container, since the annhilator value will change from operation to operation, and the insertion of explicit "zeros" could cause incorrect results. As such, no GraphBLAS operation will arbitrarily insert explicit zeros. In the GraphBLAS C++ API, each container has associated with it a scalar type, which is the type of the stored scalar values, as well as an index type, which is the type used to indicate the locations of stored values, as well as the matrix shape.

***TODO: be more specific here. Could use some help from math spec?***

### Matrices

GraphBLAS matrices are two-dimensional sparse arrays. As previously discussed, they have a scalar type, which defines the type of scalar values stored inside the matrix, as well as an index type, which is used to refer to the row and column index of each stored value. For a matrix type `A`, these types can be retrieved using the expression `grb::matrix_scalar_t<A>` to retrieve the scalar type and `grb::matrix_index_t<A>` to retrieve the index type. In the GraphBLAS C++ API, matrices are associative arrays similar to instantiations of `std::unordered_map`. They map keys, which are row and column indices, to values, which are the stored scalar values. Like the C++ standard library's

associative containers, GraphBLAS matrices are ranges, and iterating through a GraphBLAS matrix reveals a collection of tuples, with each tuple holding a key and value. The key is a tuple-like type holding the row and column indices, and the value is the stored scalar value. The row and column are provided only as const references, while the value may be mutated. Matrices also support insert, assign, and find operations to write or read to a particular location in the matrix.

### Vectors

GraphBLAS similars are analogous to matrices, except that they are only one-dimensional sparse arrays. They hold a scalar type, which defines the scalar values stored inside the vector, as well as an index type, which holds the index of an element within the vector. For a vector `v`, these types can be retrieved with the expressions `grb::vector_scalar_t<V>` and `grb::vector_index_t<V>`. Like matrices, vectors are associative arrays, except that their key values are a single index type element, and not a tuple. Like matrices, vectors are ranges, with the value type being a tuple holding the index type and scalar type, with only the scalar type being mutable if the vector is non-const. Individual elements can be modified using insert, assign, and find.

# grb::matrix

```
template <typename T,
          std::integral I = std::size_t,
          typename Hint = grb::sparse,
          typename Allocator = std::allocator<T>>
class grb::matrix;
```

## Template Parameters

`T` - type of scalar values stored in the matrix

`I` - type, satisfying `std::integral`, used to store indices in the matrix

`Hint` - one or a composition of compile-time hints that may be used to affect the backend storage type

`Allocator` - allocator type used to allocate memory for the matrix

## Member Types

| Member Type | Definition |
|---|---|
| `scalar_type` | `T`, the type of elements stored in the matrix |
| `index_type` | `I`, an integer type used to store matrix indices |
| `key_type` | A tuple-like type storing two `index_type` elements |

| Member Type | Definition |
|---|---|
| `map_type` | `scalar_type` |
| `value_type` | `grb::matrix_entry<T, I>`, a tuple-like type storing the indices and the stored element |
| `size_type` | A large unsigned integer type, usually `std::size_t` |
| `difference_type` | A large signed integer type, usually `std::ptrdiff_t` |
| `allocator_type` | `Allocator` |
| `iterator` | An iterator type fulfilling `std::forward_iterator` |
| `const_iterator` | A const iterator fulfilling `std::forward_iterator` |
| `reference` | `grb::matrix_reference<T, I>` |
| `const_reference` | `grb::matrix_reference<const T, I>` |
| `scalar_reference` | Reference to `scalar_type` |
| `const_scalar_reference` | Const reference to `scalar_type` |
| `hint_type` | `Hint` |

# Methods

| Method | Description |
|---|---|
| `(constructor)` | Constructs matrix |
| `(destructor)` | Destructs matrix |
| `operator=` | Assigns matrix |
| `size` | Number of stored elements in matrix |
| `max_size` | Returns the maximum possible number of elements |
| `empty` | Return whether the matrix is empty |
| `shape` | Return the dimensions of the matrix |
| `begin`<br>`cbegin` | Returns iterator to beginning of container |
| `end`<br>`cend` | Returns iterator to one past last element in container |
| `reshape` | Modify dimensions of the matrix |
| `clear` | Clears all elements from the container |
| `insert` | Insert elements |

| Method | Description |
|---|---|
| `insert_or_assign` | Inserts or assigns elements |
| `erase` | Erases elements |
| `find` | Finds an element |
| `at` | Access element |
| `operator[]` | Access or insert element |

## grb::matrix::matrix

```
matrix(const Allocator& alloc = Allocator());     (1)
matrix(grb::index<index_type> shape,
       const Allocator& alloc = Allocator());     (2)
matrix(const matrix& other);                      (3)
matrix(matrix&& other);                           (4)
```

Constructs new `grb::matrix` data structure.

1. Constructs an empty matrix of dimension `0 x 0`.
2. Constructs an empty matrix of dimension `shape[0] x shape[1]`.
3. Copy constructor. Constructs a matrix with the dimensions and contents of `other`.
4. Move constructor. Constructs a matrix with the dimensions and contents of `other` using move semantics. After the move, `other` is guaranteed to be empty.

### Parameters

`shape` - shape of the matrix to be constructed

`other` - another matrix to construct the new matrix from

### Complexity

1. Constant
2. Implementation defined
3. Implementation defined
4. Implementation defined

### Exceptions

Calls to `Allocator::allocate` may throw.

## grb::matrix::~matrix

```
~matrix();
```

Destroys the `grb::matrix`. The destructors of the elements are called and storage is deallocated.

### Complexity

Linear in the size of the vector.

## grb::matrix::operator=

```
matrix& operator=(const matrix& other);          (1)
matrix& operator=(matrix&& other);               (2)
```

Replaces the matrix with the contents and shape of another matrix.

1. Copy assignment operator. Modifies the shape of the matrix to be equal to that of `other` and copies the stored values to be the same as `other`.
2. Move constructor. Replaces the contents of the matrix with those of `other` using move semantics. After the move assignment operation completes, `other` is in a valid but indeterminate state.

### Parameters

`other` - another matrix to assign the matrix to

### Complexity

1. Implementation defined
2. Constant

### Exceptions

1. Calls to `Allocator::allocate` may throw.

## grb::matrix::size

```
size_type size() noexcept;
```

Returns the number of elements stored in the matrix.

### Parameters

(none)

### Return value

Number of stored elements.

### Complexity

Constant

## grb::matrix::max_size

```
size_type max_size() const noexcept;
```

Returns the maximum possible number of elements that could be stored in the matrix due to system or implementation limitations.

### Parameters

(none)

### Return value

Maximum possible number of elements.

### Complexity

Constant

## grb::matrix::empty

```
bool empty() noexcept;
```

Returns whether the matrix is empty, that is where `size() == 0`.

### Parameters

(none)

### Return value

Whether the matrix is empty.

### Complexity

Constant

## `grb::matrix::shape`

```
grb::index<I> shape() const noexcept;
```

Returns the dimensions of the matrix.

### Parameters

(none)

### Return value

Dimensions of the matrix.

### Complexity

Constant

## `grb::matrix::begin` and `grb::matrix::cbegin`

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;
```

Returns an iterator to the first element of the matrix data structure. If the matrix is empty, returns an element that compares as equal to `end()`.

### Parameters

(none)

### Return value

Iterator to the first element

### Complexity

Constant

## `grb::matrix::end` and `grb::matrix::cend`

```
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;
```

Returns an iterator to one past the last element of the matrix data structure. This element is used only to signify the end of the container, and accessing it has undefined behavior.

### Parameters

(none)

### Return value

Iterator to one past the last element.

### Complexity

Constant

## `grb::matrix::reshape`

```
void reshape(grb::index<I> shape) noexcept;
```

Reshape the matrix. That is, modify the matrix dimensions such that matrix shape is now `shape[0]` x `shape[1]`. If any nonzeros lie outside of the new matrix shape, they are removed from the matrix

### Parameters

`shape` - New matrix shape

### Return value

None

### Complexity

Implementation defined

### Exceptions

Calls to `Allocator::allocate` may throw.

## `grb::matrix::clear`

```
void clear() noexcept;
```

Clear all stored scalar values from the matrix. The matrix maintains the same shape, but after return will now have a size of 0.

### Complexity

Implementation defined

## grb::matrix::insert

```
template <typename InputIt>
void insert(InputIt first, InputIt last);                    (1)
std::pair<iterator, bool> insert(const value_type& value);   (2)
std::pair<iterator, bool> insert(value_type&& value);        (3)
```

Inserts an element or number of elements into the matrix, if the matrix doesn't already contain a stored element at the corresponding index.

1. Insert each element in the range `[first, last)` if an element does not already exist in the matrix at the corresponding index. If multiple elements in `[first, last)` have the same indices, it is undefined which element is inserted.

2. and (3) Insert the element `value` if an element does not already exist at the corresponding index in the matrix.

### Parameters

`first`, `last` - Range of elements to insert

`value` - element to insert

### Return value

1. None

2. and (3) return a `std::pair` containing an `iterator` and a `bool` value. The `bool` value indicates whether or not the insertion was successful. If the insertion was successful, the first value contains an iterator to the newly inserted element. If it was unsuccessful, it contains an iterator to the element that prevented insertion.

### Complexity

Implementation defined

## grb::matrix::insert_or_assign

```
template <class M>
std::pair<iterator, bool> insert_or_assign(key_type k, M&& obj);
```

Inserts an element into index `k` in the matrix and assigns to the scalar value if one already exists.

If an element already exists at index location `k`, assigns `std::forward<M>(obj)` to the scalar value stored at that index. If no element exists, inserts `obj` at the index as if by calling `insert` with `value_type(k, std::forward<M>(obj))`.

### Parameters

`k` - the index location to assign `obj`

`obj` - the scalar value to be assigned

### Type Requirements

`M` must fulfill `std::is_assignable<scalar_type&, M>`

### Return Value

Returns an `std::pair` with the first element holding an iterator to the newly inserted or assigned value and the second element holding a `bool` value that is `true` if the element was inserted and `false` otherwise.

### Complexity

Implementation defined.

## `grb::matrix::erase`

```
size_type erase(grb::index<I> index);
```

Erases the element stored at index `index` if one exists.

### Parameters

`index` - the index of element to erase

### Return Value

Returns the number of elements erased (0 or 1).

### Complexity

Implementation defined.

# grb::matrix::find

```
iterator find(grb::index<I> index);
const_iterator find(grb::index<I> index) const;
```

Finds the element at location (`index[0]`, `index[1]`) in the matrix, returning an iterator to the element. If no element is present at the location, returns `end()`.

## Parameters

`index` - Location of the matrix element to find

## Return value

An iterator pointing to the matrix element at the corresponding index, or `end()` if there is no element at the location.

## Complexity

Implementation defined

# grb::matrix::operator[]

```
scalar_reference operator[](grb::index<I> index);
const_scalar_reference operator[](grb::index<I> index) const;
```

Returns a reference to the scalar value stored at location (`index[0]`, `index[1]`) in the matrix. If no value is stored at the location, inserts a default constructed value at the location and returns a reference to it.

## Parameters

`index` - Location of the matrix element

## Return value

A reference to the scalar value at location (`index[0]`, `[index[1]`).

## Complexity

Implementation defined

## grb::matrix::at

```
scalar_reference at(grb::index<I> index);
const_scalar_reference at(grb::index<I> index) const;
```

Returns a reference to the scalar value stored at location (`index[0]`, `index[1]`) in the matrix. If no value is stored at the location, throws the exception `grb::out_of_range`.

### Parameters

`index` - Location of the matrix element

### Return value

A reference to the scalar value at location (`index[0]`, `[index[1]`).

### Complexity

Implementation defined

### Exceptions

If no element exists at location (`index[0]`, `index[1]`), throws the exception `grb::out_of_range`.

# grb::vector

```
template <typename T,
          std::integral I = std::size_t,
          typename Hint = grb::sparse,
          typename Allocator = std::allocator<T>>
class grb::vector;
```

## Template Parameters

`T` - type of scalar values stored in the vector

`I` - type, satisfying `std::integral`, used to store indices in the vector

`Hint` - one or a composition of compile-time hints that may be used to affect the backend storage type

`Allocator` - allocator type used to allocate memory

# Member Types

| Member Type | Definition |
|---|---|
| `scalar_type` | `T`, the type of scalar elements stored in the vector |
| `index_type` | `I`, an integer type used to store vector indices |
| `key_type` | `index_type` |
| `map_type` | `scalar_type` |
| `value_type` | `grb::vector_entry<T, I>`, a tuple-like type storing the indices and the stored element |
| `size_type` | A large unsigned integer type, usually `std::size_t` |
| `difference_type` | A large signed integer type, usually `std::ptrdiff_t` |
| `allocator_type` | `Allocator` |
| `iterator` | An iterator type fulfilling `std::forward_iterator` |
| `const_iterator` | A const iterator fulfilling `std::forward_iterator` |
| `reference` | `grb::vector_ref<T, I>` |
| `const_reference` | `grb::vector_ref<const T, I>` |
| `scalar_reference` | Reference to `scalar_type` |
| `const_scalar_reference` | Const reference to `scalar_type` |
| `hint_type` | `Hint` |

# Methods

| Method | Description |
|---|---|
| `(constructor)` | Constructs vector |
| `size` | Number of stored elements in vector |
| `max_size` | Returns the maximum possible number of elements |
| `empty` | Return whether the vector is empty |
| `shape` | Return the dimension of the vector |
| `begin`<br>`cbegin` | Returns iterator to beginning of container |
| `end`<br>`cend` | Returns iterator to one past last element in container |

| Method | Description |
| --- | --- |
| `reshape` | Modify the dimension of the vector |
| `clear` | Clears all elements from the container |
| `insert` | Insert elements |
| `insert_or_assign` | Inserts or assigns elements |
| `find` | Finds an element |
| `operator[]` | Access or insert element |

## grb::vector::vector

```
vector();                                          (1)
vector(const Allocator& alloc);                    (2)
vector(index_type shape);                          (3)
vector(index_type shape, const Allocator& alloc);  (4)
vector(const vector& other);                       (5)
vector(vector&& other);                            (6)
```

Constructs new `grb::vector` data structure.

1. and 2) Constructs an empty vector of dimension 0.
2. and 4) Constructs an empty vector of dimension `shape`.
3. Copy constructor
4. Move constructor

### Parameters

`shape` - shape of the vector to be constructed

### Complexity

1. Constant
2. Implementation defined

### Exceptions

May throw `std::bad_alloc`.

## grb::vector::size

```
size_type size() noexcept;
```

Returns the number of elements stored in the vector.

### Parameters

(none)

### Return value

Number of stored elements.

### Complexity

Constant

## grb::vector::max_size

```
size_type max_size() noexcept;
```

Returns the maximum possible number of elements that could be stored in the vector due to platform or implementation limitations.

### Parameters

(none)

### Return value

Maximum possible number of elements.

### Complexity

Constant

## grb::vector::empty

```
bool empty() noexcept;
```

Returns whether the vector is empty, that is where `size() == 0`.

### Parameters

(none)

### Return value

Whether the vector is empty.

### Complexity

Constant

## `grb::vector::shape`

```
index_type shape() const noexcept;
```

Returns the dimension of the vector.

### Parameters

(none)

### Return value

Dimension of the vector.

### Complexity

Constant

## `grb::vector::begin` and `grb::vector::cbegin`

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;
```

Returns an iterator to the first element of the vector data structure. If the vector is empty, returns an element that compares as equal to `end()`.

### Parameters

(none)

### Return value

Iterator to the first element

### Complexity

Constant

## `grb::vector::end` and `grb::vector::cend`

```
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;
```

Returns an iterator to one past the last element of the vector data structure. This element is used only to signify the end of the container, and accessing it has undefined behavior.

### Parameters

(none)

### Return value

Iterator to one past the last element.

### Complexity

Constant

## `grb::vector::reshape`

```
void reshape(index_type shape);
```

Reshape the vector. That is, modify the vector dimension such that vector is now of dimension `shape`. If any nonzeros lie outside of the new vector shape, they are removed from the vector.

### Parameters

`shape` - New vector shape

### Return value

None

### Complexity

Implementation defined

## `grb::vector::clear`

```
void clear();
```

Clear all stored scalar values from the vector. The vector maintains the same shape, but after return will now have a size of 0.

### Complexity

Implementation defined

## `grb::vector::insert`

```
template <typename InputIt>
void insert(InputIt first, InputIt last);                    (1)
std::pair<iterator, bool> insert(const value_type& value);   (2)
std::pair<iterator, bool> insert(value_type&& value);        (3)
```

Inserts an element or number of elements into the vector, if the vector doesn't already contain a stored element at the corresponding index.

1. Insert elements in the range `[first, last)` if an element does not already exist in the vector at the corresponding index. If multiple elements in `[first, last)` have the same indices, it is undefined which element is inserted.
2. and (3) Insert the element `value` if an element does not already exist at the corresponding index in the vector.

### Parameters

`first`, `last` - Range of elements to insert

`value` - element to insert

### Return value

1. None
2. and (3) return a `std::pair` containing an `iterator` and a `bool` value. The `bool` value indicates whether or not the insertion was successful. If the insertion was successful, the first value contains an iterator to the newly inserted element. If it was unsuccessful, it contains an iterator to the element that prevented insertion.

### Complexity

Implementation defined

## grb::vector::insert_or_assign

```
template <class M>
std::pair<iterator, bool> insert_or_assign(key_type k, M&& obj);
```

Inserts an element into index `k` in the vector and assigns to the scalar value if one already exists.

If an element already exists at index location `k`, assigns `std::forward<M>(obj)` to the scalar value stored at that index. If no element exists, inserts `obj` at the index as if by calling `insert` with `value_type(k, std::forward<M>(obj))`.

### Parameters

`k` - the index location to assign `obj`

`obj` - the scalar value to be assigned

### Type Requirements

`M` must fulfill `std::is_assignable<scalar_type&, M>`

### Return Value

Returns an `std::pair` with the first element holding an iterator to the newly inserted or assigned value and the second element holding a `bool` value that is `true` if the element was inserted and `false` otherwise.

### Complexity

Implementation defined.

## grb::vector::erase

```
size_type erase(index_type index);
```

Erases the element stored at index `index` if one exists.

### Parameters

`index` - the index of the scalar value to erase

### Return Value

Returns the number of elements erased (0 or 1).

### Complexity

Implementation defined.

## grb::vector::find

```
iterator find(index_type index);
const_iterator find(index_type index) const;
```

Finds the element at location `index` in the vector, returning an iterator to the element. If no element is present at the location, returns `end()`.

### Parameters

`index` - Location of the vector element to find

### Return value

An iterator pointing to the vector element at the corresponding index, or `end()` if there is no element at the location.

### Complexity

Implementation defined

## grb::vector::operator[]

```
scalar_reference operator[](index_type index);
const_scalar_reference operator[](index_type index) const;
```

Returns a reference to the scalar value stored at location `index` in the vector. If no value is stored at the location, inserts a default constructed value at the location and returns a reference to it.

### Parameters

`index` - Location of the vector element

### Return value

A reference to the scalar value at location `index`.

### Complexity

Implementation defined

## grb::vector::at

```
scalar_reference at(index_type index);
const_scalar_reference at(index_type index) const;
```

Returns a reference to the scalar value stored at location `index` in the vector. If no value is stored at the location, throws the exception `grb::out_of_range`.

### Parameters

`index` - Location of the vector element

### Return value

A reference to the scalar value at location `index`.

### Complexity

Implementation defined

### Exceptions

If no element exists at location `index`, throws the exception `grb::out_of_range`.

# grb::index

```
template <std::integral T>
struct grb::index;
```

`grb::index` is a tuple-like type used to store matrix indices and dimensions.

## Template Parameters

`T` - type, satisfying `std::integral`, used to store indices

## Member Types

| Member Type | Definition |
|---|---|
| `index_type` | `T` |
| `first_type` | `T` |
| `second_type` | `T` |

## Member Objects

| Member Name | Type |
|---|---|
| `first` | `T` |
| `second` | `T` |

## Methods

| Method | Description |
|---|---|
| `(constructor)` | Constructs index |
| `operator[]` | Get one of the indices |
| `get` | Get one of the indices |

# Sparse Storage Format Hints

This section defines compile-time hints that can be used to direct the storage format used to organize data inside GraphBLAS objects. Storage hints can be used as optional template arguments to `grb::matrix`, `grb::vector`, as well as some API functions that return a matrix or vector. These compile-time hints provide a mechanism for users to suggest to the GraphBLAS implementation which storage format might be most appropriate for a particular matrix. However, these compile-time hints are indeed *hints* to the implementation and do not enforce any change in semantics to a matrix or vector data structure and do not require it to use any particular storage format. GraphBLAS implementations are encouraged to document which storage formats they support, as well as how different hints will affect which storage formats are used.

### Example

```
#include <grb/grb.hpp>

int main(int argc, char** argv) {
  // Create a matrix with the `dense` compile-time hint.
  // The implementation may choose to employ a dense storage format.
  grb::matrix<float, std::size_t, grb::dense> x({10, 10});

  // Create a matrix with the compile-time hint compose<sparse, row>.
  // A GraphBLAS implementation may choose to employ a compressed sparse row
  // storage format.
  grb::matrix<float, std::size_t, grb::compose<grb::sparse, grb::row>> y({10, 10});
```

```
  // No change in data structure semantics is implied by a hint.
  x[{2, 3}] = 12;
  x[{3, 6}] = 12;

  y[{6, 7}] = 1;
  y[{6, 7}] = 2.5;

  grb::print(x);
  grb::print(y);

  return 0;
}
```

## grb::sparse

```
using sparse = /* undefined */;
```

Compile-time hint to suggest to the GraphBLAS implementation that a sparse or compressed storage format is appropriate.

## grb::dense

```
using dense = /* undefined */;
```

Compile-time hint to suggest to the GraphBLAS implementation that a dense storage format is appropriate.

## grb::row

```
using row = /* undefined */;
```

Compile-time hint to suggest to the GraphBLAS implementation that a row-major storage format is appropriate.

## grb::column

```
using column = /* undefined */;
```

Compile-time hint to suggest to the GraphBLAS implementation that a column-major storage format is appropriate.

### grb::compose

```
template <typename... Hints>
using compose = /* undefined */;
```

Type alias used to combine multiple compile-time hints together.

# Views

## grb::views::transpose

```
namespace views {
  inline constexpr /* unspecified */ transpose = /* unspecified */;
}
```

#### *Call signature*

```
template <MatrixRange M>
MatrixRange auto transpose(M&& matrix);
```

### Parameters

`matrix` - a GraphBLAS matrix or matrix view

#### Type Requirements

- `M` must meet the requirements of `MatrixRange`

### Return Value

Returns a matrix view that is equal to the transpose of `matrix`, satisfying `MatrixRange`.

If the value returned by `transpose` outlives the lifetime of `matrix`, then the behavior is undefined.

## grb::views::transform

```
namespace views {
  inline constexpr /* unspecified */ transform = /* unspecified */;
}
```

#### *Call signature*

```
template <MatrixRange M, std::copy_constructible Fn>
MatrixRange auto transform (M&& matrix, Fn&& fn);              (1)

template <VectorRange V, std::copy_constructible Fn>
VectorRange auto transform(V&& vector, Fn&& fn);               (2)
```

1. `fn` must accept an argument of type `std::ranges::range_value_t<M>` and return a value of some type. The return type of `fn` will be the scalar type of the transform view.
2. `fn` must accept an argument of type `std::ranges::range_value_t<V>` and return a value of some type. The return type of `fn` will be the scalar type of the transform view.

## Parameters

`matrix` - a GraphBLAS matrix or matrix view satisfying `MatrixRange`

`vector` - a GraphBLAS vector or vector view satisfying `VectorRange`

`fn` - function used to transform matrix values element-wise

### Type Requirements

- `M` must meet the requirements of `MatrixRange`
- `V` must meet the requirements of `VectorRange`

## Return Value

1. Returns a view of `matrix` satisfying `MatrixRange`. The stored scalar values will correspond to every stored scalar value in `matrix` transformed using the function `fn`.
2. Returns a view of `vector` satisfying `VectorRange`. The stored scalar values will correspond to every stored scalar value in `vector` transformed using the function `fn`.

## Example

```
#include <grb/grb.hpp>

int main(int argc, char** argv) {
  grb::matrix<float> a({10, 10});

  a[{1, 2}] = 12.0f;
  a[{3, 4}] = 123.2f;
  a[{7, 8}] = 8.0f;;

  grb::print(a, "original matrix");
```

```
  // Transform each scalar value based on its
  // row and column index.
  auto idx_view = grb::views::transform(a, [](auto&& entry) {
                                          auto&& [idx, v] = entry;
                                          auto&& [i, j] = idx;
                                          return i + j;
                                        });

  grb::print(idx_view, "index transformed view");

  return 0;
}
```

# grb::structure

```
template <MatrixRange M>
grb::structure_view<M> structure(M&& matrix);                (1)

template <VectorRange V>
grb::structure_view<V> structure(V&& vector);                (2)
```

## Parameters

`matrix` - a GraphBLAS matrix or matrix view

`vector` - a GraphBLAS vector or vector view

### Type Requirements

- `matrix` must meet the requirements of `MatrixRange`
- `vector` must meet the requirements of `VectorRange`

## Return Value

1. Returns a view of the matrix

### Possible Implementation

```
inline constexpr bool always_true = [](auto&&) { return true; };

template <typename O>
using structure_view = grb::transform_view<O, decltype(always_true)>;

template <MatrixRange M>
```

```
grb::structure_view<M> structure(M&& matrix) {
  return grb::transform(std::forward<M>(matrix), always_true);
}

template <VectorRange V>
grb::structure_view<V> structure(V&& vector) {
  return grb::transform(std::forward<V>(vector), always_true);
}
```

# grb::filter

```
template <MatrixRange M, typename Fn>
grb::filter_view<M> filter(M&& matrix, Fn&& fn);              (1)

template <VectorRange V, typename Fn>
grb::filter_view<V> filter(V&& vector, Fn&& fn);              (2)
```

## Parameters

`matrix` - a GraphBLAS matrix

`vector` - a GraphBLAS vector

`fn` - a function determining which elements should be filtered out

### Type Requirements

- `M` must meet the requirements of `MaskMatrixRange`
- `V` must meet the requirements of `MaskVectorRange`
- `fn` must accept an argument of type `std::ranges::value_type_t<M>` (1) or `std::ranges::value_type_t<V>` (2) and return a value of type `bool`.

## Return Value

Returns a filtered view of the matrix `matrix` or vector `vector`. The return value fulfills the requirements of `MatrixRange` (1) or `VectorRange` (2) . The return value has the same shape as the input object, but without elements for which `fn(e)` evaluates to false.

# grb::complement

```
template <MaskMatrixRange M>
grb::complement_view<M> complement(M&& mask);                 (1)
```

```
template <MaskVectorRange V>
grb::complement_view<V> complement(V&& mask);                    (2)
```

### Parameters

`mask` - a GraphBLAS mask

#### Type Requirements

- `M` must meet the requirements of `MaskMatrixRange`
- `V` must meet the requirements of `MaskVectorRange`

### Return Value

*TODO: Scott, Jose, is this the behavior we want?* Returns the complement view of a matrix or vector mask. At every index in `mask` with no stored value or with a scalar value equal to `false` when converted to `bool`, the returned view has a stored value with the scalar value `true`. At every stored value in `mask` with a scalar value equal to `true` when converted to `bool`, the return view has no stored value.

1. Returns a matrix view satisfying the requirements `MaskMatrixRange`.
2. Returns a vector view satisfying the requirements `MaskVectorRange`.

## grb::mask

```
template <MatrixRange Matrix, MaskMatrixRange M>
grb::masked_view<Matrix, M> mask(Matrix&& matrix, M&& mask);                    (1)

template <VectorRange Vector, MaskVectorRange M>
grb::masked_view<Vector, M> mask(Vector&& vector, M&& mask);                    (2)
```

### Parameters

`matrix` - a GraphBLAS matrix or vector

`mask` - a GraphBLAS mask with the same shape as `matrix` (1) or `vector` (2)

#### Type Requirements

- `Matrix` must meet the requirements of `MatrixRange`
- `Vector` must meet the requirements of `VectorRange`
- `M` must meet the requirements of `MaskMatrixRange` (1) or `MaskVectorRange` (2)

### Return Value

Returns a view of the matrix `matrix` (1) or vector `vector` that has been masked using `mask`. Any value in `matrix` for which `mask` does not have an entry at the corresponding location or for which `mask` has a value equal to `false` when converted to `bool` will not be visible in the returned mask view.

1. Returns a matrix view satisfying the requirements `MatrixRange`. Any values in `matrix` for which `grb::find(mask, index) == grb::end(mask) || bool(*grb::find(mask, index)) == false` will be masked out of the returned matrix view.

### Exceptions

The exception `grb::invalid_argument` is thrown if the `mask` does not have the same shape as `matrix` (1) or `vector` (2).

## grb::submatrix_view

```
template <MatrixRange M>
grb::submatrix_view<M> submatrix(M&& matrix, grb::index<I> rows, grb::index<I> cols);
```

### Parameters

`matrix` - a GraphBLAS matrix

`rows` - the span of rows in the submatrix view

`cols` - the span of columns in the submatrix view

### Return Value

Returns a view of the submatrix of GraphBLAS matrix `matrix` formed by the intersection of rows `rows[0]` to `rows[1]` and columns `cols[0]` to `cols[1]`. The returned matrix range has shape `grb::min(rows[1], grb::shape(matrix)[0]) - grb::max(rows[0], 0)` by `grb::min(cols[1], grb::shape(matrix)[1]) - grb::max(cols[0], 0)` and contains all stored values for whose index the expression `index[0] >= rows[0] && index[0] < rows[1] && index[1] >= cols[0] && index[1] < cols[1]` holds true.

### Exceptions

Returns the exception `grb::invalid_argument` if `rows[0] > rows[1]` or `cols[0] > cols[1]`.

# Pre-Defined Operators

The GraphBLAS C++ API defines a number of binary and unary operators. Operators are defined as function objects and typically perform arithmetic or logical operations like addition, multiplication, negation, and logical and.

**TODO: Discuss Unary Operators**

They work the same way as binary operators, but are quite a bit simpler, since there is only one value to operate upon.

# Binary Operator Template Parameters

GraphBLAS's binary operators allow flexiblility in terms of whether all, some, or none of the types are explicitly declared. If one or more types are not explicitly declared, they are deduced at the callsite where the operator is invoked. For each binary operator `op`, there are three partial specializations defined:

1. `op<T, U, V>`, where the input arguments are explicitly specified to be `T` and `U` and the return value is `V`.
2. `op<T, U, void>`, where the input arguments are explicitly specified to be `T` and `U`, and the return value is deduced.
3. `op<void, void, void>`, where the inputs arguments and return type are all deduced upon invocation.

All binary operators have the following default template parameters:

```
template <typename T = void, typename U = T, typename V = void>
struct op;
```

This means the user can use a binary operator with zero, one, two, or three template parameters explicitly specified. The types of the input arguments and return value will be explicitly defined or reduced accordingly.

### No Types Specified

If a user specifies no template parameters when using a binary operator, the result is a function object in which the input argument and return value types are all deduced at the callsite.

```
#include <grb/grb.hpp>
#include <iostream>
int main(int argc, char** argv) {
  // No types are explicitly specified.
  // We can use this operator with values of
  // any types for which an operator+() is defined.
  grb::plus<> p1;

  std::size_t a = 1024;
```

```
  std::uint8_t b = 12;

  // `std::size_t` and `std::uint8_t` deduced as types of input arguments.
  // `std::size_t` will be deduced as type of return value.
  // std::size_t + std::uint8_t -> std::size_t
  auto c = p1(a, b);

  return 0;
}
```

### One Type Specified

If a user specifies one template parameters when using a binary operator, the result is a function object in which the the two input arguments are of the type specified, and the return value type is deduced.

```
#include <grb/grb.hpp>
#include <iostream>
int main(int argc, char** argv) {
  // Explicitly specify `int` as the type of both input values.
  // Return value type will be deduced as `int`.
  grb::plus<int> p1;

  int a = 1024;
  int b = 12;

  // int + int -> int
  int c = p1(a, b);

  return 0;
}
```

### Two Types Specified

If a user specifies two template parameters when using a binary operator, the result is a function object in which the the two input arguments are of the types specified, and the return value type is deduced.

```
#include <grb/grb.hpp>
#include <iostream>
int main(int argc, char** argv) {
  // Explicitly specify `std::size_t` for the lefthand argument
  // and `std::uint8_t` for the righthand argument.
  // Return value type will be deduced as `std::size_t`.
  grb::plus<std::size_t, std::uint8_t> p1;

  std::size_t a = 1024;
  std::uint8_t b = 12;
```

```
  // std::size_t + std::uint8_t -> std::size_t
  std::size_t c = p1(a, b);

  return 0;
}
```

### Three Types Specified

If a user specifies all three template parameters when using a binary operator, both of the input argument types as well as the return value type of the function are explicitly specified.

```
#include <grb/grb.hpp>
#include <iostream>
int main(int argc, char** argv) {
  // Explicitly specify `std::size_t` for the lefthand argument
  // and `std::uint8_t` for the righthand argument.
  // Return value type explicitly specified as `int`.
  grb::plus<std::size_t, std::uint8_t, int> p1;

  std::size_t a = 1024;
  std::uint8_t b = 12;

  // std::size_t + std::uint8_t -> std::size_t
  std::size_t c =  p1(a, b);

  return 0;
}
```

## grb::plus

```
template <typename T = void, typename U = T, typename V = void>
struct plus;
```

`grb::plus` is a binary operator. It forms a monoid on arithmetic types.

### Specializations

| Specialization | Description |
|---|---|
| `plus<void, void, void>` | deduces argument and return types |
| `plus<T, U, void>` | deduces return type based on `T` and `U` |

## Template Parameters

`T` - Type of the left hand argument of the operator

`U` - Type of the right hand argument of the operator

`V` - Return type of the operator

## Methods

| Method | Description |
|---|---|
| `operator()` | returns the sum of the two arguments |

`grb::plus::operator()`

```
constexpr V operator()( const T& lhs, const U& rhs ) const;
```

Returns the sum of `lhs` and `rhs`.

### Parameters

`lhs`, `rhs` - values to sum

### Return Value

The result of `lhs + rhs`.

### Exceptions

May throw exceptions if the underlying `operator+()` operation throws exceptions

### Monoid Traits

`grb::plus` forms a monoid on any arithmetic type `A` with the identity value `0`, as long as `T`, `U`, and `V` are equal to void or `A`.

## Specialization Details

`grb::plus<void, void, void>`

```
template <>
struct plus<void, void, void>;
```

Version of `grb::plus` with both arguments and return types deduced.

`grb::plus::operator()`

```
template <typename T, typename U>
constexpr auto operator()(T&& lhs, U&& rhs) const
  -> decltype(std::forward<T>(lhs) + std::forward<U>(rhs));
```

`grb::plus<T, U, void>`

```
template <typename T = void, typename U = T>
struct plus<T, U, void>;
```

Version of `grb::plus` with explicit types for the arguments, but return type deduced.

`grb::plus::operator()`

```
constexpr auto operator()(const T& lhs, const U& rhs) const
  -> decltype(lhs + rhs);
```

## `grb::multiplies`

```
template <typename T = void, typename U = T, typename V = void>
struct multiplies;
```

`grb::multiplies` is a binary operator. It forms a monoid on arithmetic types.

### Specializations

| Specialization | Description |
|---|---|
| `multiplies<void, void, void>` | deduces argument and return types |
| `multiplies<T, U, void>` | deduces return type based on `T` and `U` |

### Template Parameters

`T` - Type of the left hand argument of the operator

`U` - Type of the right hand argument of the operator

`V` - Return type of the operator

### Methods

| Method | Description |
|---|---|
| `operator()` | returns the product of the two arguments |

`grb::multiplies::operator()`

```
constexpr V operator()( const T& lhs, const U& rhs ) const;
```

Returns the product of `lhs` and `rhs`.

### Parameters

`lhs`, `rhs` - values to multiply

### Return Value

The result of `lhs * rhs`.

### Exceptions

May throw exceptions if the underlying `operator*()` operation throws exceptions

### Monoid Traits

`grb::multiplies` forms a monoid on any arithmetic type `A` with the identity value `1`, as long as `T`, `U`, and `V` are equal to void or `A`.

### Specialization Details

`grb::multiplies<void, void, void>`

```
template <>
struct multiplies<void, void, void>;
```

Version of `grb::multiplies` with both arguments and return types deduced.

`grb::multiplies::operator()`

```
template <typename T, typename U>
constexpr auto operator()(T&& lhs, U&& rhs) const
  -> decltype(std::forward<T>(lhs) * std::forward<U>(rhs));
```

`grb::multiplies<T, U, void>`

```
template <typename T = void, typename U = T>
struct multiplies<T, U, void>;
```

Version of `grb::multiplies` with explicit types for the arguments, but return type deduced.

`grb::multiplies::operator()`

```
constexpr auto operator()(const T& lhs, const U& rhs) const
  -> decltype(lhs * rhs);
```

## grb::minus

```
template <typename T = void, typename U = T, typename V = void>
struct minus;
```

`grb::minus` is a binary operator that subtracts two values.

### Specializations

| Specialization | Description |
|---|---|
| `minus<void, void, void>` | deduces argument and return types |
| `minus<T, U, void>` | deduces return type based on `T` and `U` |

### Template Parameters

`T` - Type of the left hand argument of the operator

`U` - Type of the right hand argument of the operator

`V` - Return type of the operator

### Methods

| Method | Description |
|---|---|
| `operator()` | returns the difference of the two arguments |

`grb::minus::operator()`

```
constexpr V operator()( const T& lhs, const U& rhs ) const;
```

Returns the difference of `lhs` and `rhs`.

#### Parameters

`lhs`, `rhs` - values to subtract

#### Return Value

The result of `lhs - rhs`.

### Exceptions

May throw exceptions if the underlying `operator-()` operation throws exceptions

### Monoid Traits

`grb::minus` does not form a monoid for arithmetic types.

## Specialization Details

`grb::minus<void, void, void>`

```
template <>
struct minus<void, void, void>;
```

Version of `grb::minus` with both arguments and return types deduced.

`grb::minus::operator()`

```
template <typename T, typename U>
constexpr auto operator()(T&& lhs, U&& rhs) const
  -> decltype(std::forward<T>(lhs) - std::forward<U>(rhs));
```

`grb::minus<T, U, void>`

```
template <typename T = void, typename U = T>
struct minus<T, U, void>;
```

Version of `grb::minus` with explicit types for the arguments, but return type deduced.

`grb::minus::operator()`

```
constexpr auto operator()(const T& lhs, const U& rhs) const
  -> decltype(lhs - rhs);
```

# `grb::divides`

```
template <typename T = void, typename U = T, typename V = void>
struct divides;
```

`grb::divides` is a binary operator that divides two values.

## Specializations

| Specialization | Description |
|---|---|
| `divides<void, void, void>` | deduces argument and return types |
| `divides<T, U, void>` | deduces return type based on `T` and `U` |

## Template Parameters

`T` - Type of the left hand argument of the operator

`U` - Type of the right hand argument of the operator

`V` - Return type of the operator

## Methods

| Method | Description |
|---|---|
| `operator()` | returns the difference of the two arguments |

`grb::divides::operator()`

```
constexpr V operator()( const T& lhs, const U& rhs ) const;
```

Returns the difference of `lhs` and `rhs`.

### Parameters

`lhs`, `rhs` - values to divide

### Return Value

The result of `lhs / rhs`.

### Exceptions

May throw exceptions if the underlying `operator/()` operation throws exceptions

### Monoid Traits

`grb::divides` does not form a monoid for arithmetic types.

### Specialization Details

`grb::divides<void, void, void>`

```
template <>
struct divides<void, void, void>;
```

Version of `grb::divides` with both arguments and return types deduced.

`grb::divides::operator()`

```
template <typename T, typename U>
constexpr auto operator()(T&& lhs, U&& rhs) const
  -> decltype(std::forward<T>(lhs) / std::forward<U>(rhs));
```

`grb::divides<T, U, void>`

```
template <typename T = void, typename U = T>
struct divides<T, U, void>;
```

Version of `grb::divides` with explicit types for the arguments, but return type deduced.

#### *grb::divides::operator()*

```
constexpr auto operator()(const T& lhs, const U& rhs) const
  -> decltype(lhs / rhs);
```

## `grb::max`

```
template <typename T = void, typename U = T, typename V = void>
struct max;
```

`grb::max` is a binary operator that returns the larger of two arguments. When both input arguments have integral types, `std::cmp_less` is used for comparison. When one or both arguments have non-integral types, `operator<` is used for comparison. It forms a monoid on arithmetic types.

### Specializations

| Specialization | Description |
|---|---|
| `max<void, void, void>` | deduces argument and return types |
| `max<T, U, void>` | deduces return type based on `T` and `U` |

### Template Parameters

`T` - Type of the left hand argument of the operator

`U` - Type of the right hand argument of the operator

`V` - Return type of the operator

## Methods

| Method | Description |
|---|---|
| `operator()` | returns the larger of two arguments |

`grb::max::operator()`

```
constexpr V operator()( const T& lhs, const U& rhs ) const;
```

Returns the larger of `lhs` and `rhs`.

### Parameters

`lhs`, `rhs` - values to take the max of

### Return Value

The larger of `lhs` and `rhs`.

### Exceptions

May throw exceptions if the underlying `operator<()` operation throws exceptions

### Monoid Traits

`grb::max` forms a monoid on any arithmetic type `A` with the identity value `std::min(std::numeric_limits<A>::lowest(), -std::numeric_limits<A>::infinity())`, as long as `T`, `U`, and `V` are equal to void or `A`.

## Specialization Details

`grb::max<void, void, void>`

```
template <>
struct max<void, void, void>;
```

Version of `grb::max` with both arguments and return types deduced.

`grb::max::operator()`

```
template <std::integral T, std::integral U>
constexpr auto operator()(const T& a, const U& b) const
```

```
  -> std::conditional_t<
      std::cmp_less(std::numeric_limits<T>::max(),
                    std::numeric_limits<U>::max()),
      U, T
     >;

template <typename T, typename U>
constexpr auto operator()(const T& a, const U& b) const
  -> std::conditional_t<
      std::numeric_limits<T>::max() < std::numeric_limits<U>::max(),
      U, T
     >
requires(!(std::is_integral_v<T> && std::is_integral_v<U>));
```

```
grb::max<T, U, void>
```

```
template <typename T = void, typename U = T>
struct max<T, U, void>;
```

Version of `grb::max` with explicit types for the arguments, but return type deduced.

```
grb::max::operator()
```

```
constexpr auto operator()(const T& a, const U& b) const
  -> std::conditional_t<
      std::cmp_less(std::numeric_limits<T>::max(),
                    std::numeric_limits<U>::max()),
      U, T
     >;

constexpr auto operator()(const T& a, const U& b) const
  -> std::conditional_t<
      std::numeric_limits<T>::max() < std::numeric_limits<U>::max(),
      U, T
     >
requires(!(std::is_integral_v<T> && std::is_integral_v<U>));
```

## grb::min

```
template <typename T = void, typename U = T, typename V = void>
struct min;
```

`grb::min` is a binary operator that returns the smaller of two arguments. using `operator<` for comparison. When both input arguments have integral types, `std::cmp_less` is used for comparison. When one or both arguments have non-integral types, `operator<` is used for comparison. It forms a

monoid on arithmetic types.

## Specializations

| Specialization | Description |
|---|---|
| `min<void, void, void>` | deduces argument and return types |
| `min<T, U, void>` | deduces return type based on `T` and `U` |

## Template Parameters

`T` - Type of the left hand argument of the operator

`U` - Type of the right hand argument of the operator

`V` - Return type of the operator

## Methods

| Method | Description |
|---|---|
| `operator()` | returns the smaller of two arguments |

`grb::min::operator()`

```
constexpr V operator()( const T& lhs, const U& rhs ) const;
```

Returns the smaller of `lhs` and `rhs`.

### Parameters

`lhs`, `rhs` - values to take the min of

### Return Value

The smaller of `lhs` and `rhs`.

### Exceptions

May throw exceptions if the underlying `operator<()` operation throws exceptions

### Monoid Traits

`grb::min` forms a monoid on any arithmetic type `A` with the identity value `std::max(std::numeric_limits<A>::max(), std::numeric_limits<A>::infinity())`, as long as `T`, `U`, and `V` are equal to void or `A`.

## Specialization Details

grb::min<void, void, void>

```
template <>
struct min<void, void, void>;
```

Version of grb::min with both arguments and return types deduced.

grb::min::operator()

```
template <std::integral T, std::integral U>
constexpr auto operator()(const T& a, const U& b) const
  -> std::conditional_t<
       std::cmp_less(std::numeric_limits<U>::lowest(),
                     std::numeric_limits<T>::lowest()),
       U, T
     >;

template <typename T, typename U>
constexpr auto operator()(const T& a, const U& b) const
  -> std::conditional_t<
       std::numeric_limits<U>::lowest() < std::numeric_limits<T>::lowest(),
       U, T
     >
requires(!(std::is_integral_v<T> && std::is_integral_v<U>));
```

grb::min<T, U, void>

```
template <typename T = void, typename U = T>
struct min<T, U, void>;
```

Version of grb::min with explicit types for the arguments, but return type deduced.

grb::min::operator()

```
constexpr auto operator()(const T& a, const U& b) const
  -> std::conditional_t<
       std::cmp_less(std::numeric_limits<U>::lowest(),
                     std::numeric_limits<T>::lowest()),
       U, T
     >;

constexpr auto operator()(const T& a, const U& b) const
  -> std::conditional_t<
       std::numeric_limits<U>::lowest() < std::numeric_limits<T>::lowest(),
```

```
      U, T
    >
requires(!(std::is_integral_v<T> && std::is_integral_v<U>));
```

# Utilities

*TODO: need "disablers" for all these CPOs. Also, "converted to its decayed type?"*

## grb::get

```
inline constexpr /* unspecified */ get = /* unspecified */;
```

### Call Signature

```
template <std::size_t I, typename T>
requires /* see below */
constexpr std::tuple_element<I, T> get(T&& tuple);
```

*TODO: we need someone who understands CPOs a little better to understand this.*

Returns the `I'th` element stored in the tuple-like object `tuple`.

A call to `grb::get` is expression-equivalent to:

1. `tuple.get<I>()`, if that expression is valid.
2. Otherwise, any calls to `get(tuple)` found through argument-dependent lookup.
3. Otherwise, `std::get<I>(tuple)`.

In all other cases, a call to `grb::get` is ill-formed.

## grb::size

```
template <typename T>
inline constexpr /* unspecified */ size = std::ranges::size;
```

### Call Signature

```
template <typename T>
constexpr auto size(T&& t);
```

Returns the number of stored values in a GraphBLAS matrix, vector, or other range. Whenever `grb::size(e)` is valid for an expression `e`, the return type is integer-like.

A call to `grb::size(t)` is expression-equivalent to:

1. `t.size()`, if that expression is valid.
2. Otherwise, any calls to `size(t)`, found through argument-dependent lookup.

In all other cases, a call to `grb::size` is ill-formed.

## grb::shape

```
template <typename T>
inline constexpr /* unspecified */ shape = /* unspecified */;
```

### Call Signature

```
template <typename T>
requires /* see below */
constexpr auto shape(T&& t);
```

Returns the shape of a GraphBLAS object. If `T` fulfills the requirements of `MatrixRange`, then the return type will be a tuple-like type storing two integer-like values, fulfilling the requirements of `TupleLike<I, I>`, where `I` is the index type of `T`. If `T` fulfills the requirements of `VectorRange`, then the return type will be `I`, where `I` is the index type of `T`.

A call to `grb::shape` is expression-equivalent to:

1. `t.shape()` if that expression is valid.
2. Otherwise, `shape(t)`, if that expression is valid, including any declarations of `shape` found through argument-dependent lookup.

In all other cases, a call to `grb::shape` is ill-formed.

## grb::find

```
template <typename T>
inline constexpr /* unspecified */ find = /* unspecified */;
```

### Call Signature

```
template <typename R, typename T>
requires /* see below */
constexpr auto find(R&& r, T&& t);
```

Return an iterator to the stored value in the GraphBLAS matrix or vector `r` at index location `t`.

A call to `grb::find` is expression-equivalent to:

1. `r.find(t)` if that expression is valid.
2. Otherwise, any calls to `find(r, t)` found through argument-dependent lookup.
3. Otherwise, any calls to `std::find(r, t)`, if that expression is valid.

In all other cases, a call to `grb::find` is ill-formed.

## grb::insert

```
template <typename T>
inline constexpr /* unspecified */ insert = /* unspecified */;
```

### Call Signature

```
template <typename R, typename T>
requires /* see below */
constexpr auto insert(R&& r, T&& t);
```

Attempt to insert the value `t` into the GraphBLAS matrix or vector `r`.

A call to `grb::insert` is expression-equivalent to:

1. `r.insert(t)` if that expression is valid.
2. Otherwise, any calls to `insert(r, t)` found through argument-dependent lookup.

## scalar_result_type_t

```
template <typename A, typename B, typename Combine>
using scalar_result_type_t = std::declval<Combine>()(
                                std::declval<scalar_type_t<A>>(),
                                std::declval<scalar_type_t<B>>());
```

The type of the scalar elements produced by combining GraphBLAS objects of type `A` and `B` elementwise using a binary operator of type `Combine`.

## multiply_result_t

```
template <typename A, typename B, typename Reduce, typename Combine, typename Mask>
using multiply_result_t = /* undefined */;
```

The type returned when multiply is called on GraphBLAS matrix or vector ranges of type `A` and `B` using binary operators `Reduce` and `Combine` with mask `Mask`.

The scalar type of the returned matrix will be `combine_result_t<A, B, Combine>`, and the index type will be `grb::container_index_t<A>` or `grb::container_index_t<B>`, whichever is able to store the larger value.

### combine_result_t

```
template <typename A, typename B, typename Combine>
using combine_result_t = std::invoke_result_t<Combine, A, B>;
```

The type returned when the binary operator `Combine` is called with an element of the scalar type of `A` as the first argument and an element of the scalar type of `B` as the second argument. `A` and `B` must be GraphBLAS matrix or vector objects.

### ewise_union_result_t

```
template <typename A, typename B, typename Combine, typename Mask>
using ewise_union_result_t = /* undefined */;
```

The type returned when `ewise_union` is called on the GraphBLAS matrix or vector ranges of type `A` and `B` using the binary operator `Combine`.

### ewise_intersection_result_t

```
template <typename A, typename B, typename Combine, typename Mask>
using ewise_intersection_result_t = /* undefined */;
```

The type returned when `ewise_intersection` is called on the GraphBLAS matrix or vector ranges of type `A` and `B` using the binary operator `Combine`.

### read_matrix

```
template <typename T,
          std::integral I = std::size_t,
          typename Hint = grb::sparse,
          typename Allocator = std::allocator<T>>
grb::matrix<T, I, Hint, Allocator>
read_matrix(std::string file_path);
```

Constructs a new GraphBLAS matrix based on the contents of a file.

## Template Parameters

`T` - type of scalar values stored in matrix

`I` - type of matrix indices

`Hint` - compile-time hint for backend storage format.

`Allocator` - allocator type

### Parameters

`file_path` - string holding the file path of the matrix to be read

### Return Value

Returns a GraphBLAS matrix whose dimensions and contents correspond to the file located at `file_path`.

### Complexity

Complexity is implementation-defined.

### Exceptions

- If the file at `file_path` cannot be read, the exception `grb::matrix_io::file_error` is thrown.
- If the file at `file_path` is recognized as an unsupported format, the exception `grb::matrix_io::unsupported_file_format` is thrown.
- Calls to `Allocator::allocate` may throw.

### Notes

GraphBLAS implementations must define the file formats that they support. Implementations are strongly encouraged to support the MatrixMarket format.

### Example

# Concepts

## Binary Operator

Binary operators are function objects that are invocable with two arguments, returning a single value. Binary operators can be callables defined by a GraphBLAS library, the C++ Standard Library, or application developers. Binary operators may be callable with a variety of different types, or only with elements of a single type. We say a binary operator is valid for a particular operation `T x U -> V` if it is callable with arguments of type `T` and `U` and returns a value convertible to type `V`.

### Requirements

1. Callable of type `Fn` can be invoked with two arguments of type `T` and `U`.

2. The return value is convertible to type `V`.

### Concept

```
template <typename Fn, typename T, typename U = T, typename V = grb::any>
concept BinaryOperator = requires(Fn fn, T t, U u) {
                           {fn(t, u)} -> std::convertible_to<V>;
                         };
```

## Remarks

The last two arguments are defaulted unless explicit template parameters are provided. The second parameter to the concept, `U`, which is the right-hand side input to the binary operator, defaults to `T`. The final parameter, `V`, which represents the output of the binary operator, defaults to `grb::any`, meaning that the return value may have any type.

## Example

```
// Constrain `Fn` to require a binary operator that accepts two integers
// as arguments and returns a value of any type.
template <BinaryOperator<int> Fn>
auto apply(Fn&& fn, int a, int b) {
  return fn(a, b);
}

// The following code will result in an error, since the function passed
// does not fulfill the BinaryOperator<int> requirement, as it cannot accept
// integers.
void test() {
  auto fn = [](std::string a, std::string b) { return a.size() + b.size(); };
  apply(fn, 12, 13);
}
```

# Monoid

*GraphBLAS monoids* are commutative monoids. Throughout this specification, they are referred to simply as monoids. They are binary operators that have special properties when applied to elements of some type `T`. We say that a function object `Fn` forms on a monoid on type `T` if the following requirements are met.

## Requirements

1. Callable of type `Fn` fulfills the concept `BinaryOperator<T, T, T>` for some type `T`.
2. The operation is associative and commutative for type `T`.

3. The operation has an identity element for type `T`, accessible using `grb::monoid_traits<Fn, T>::identity()`.

### Concept

```
template <typename Fn, typename T>
concept Monoid = BinaryOperator<Fn, T, T, T> &&
                 requires { {grb::monoid_traits<Fn, T>::identity()} -> std::same_as<T>;
        };
```

# Tuple-Like Type

Tuple-like types are types that, similar to instantiations of `std::tuple` or `std::pair`, store multiple values. The number of values stored in the tuple-like type, as well as the type of each value, are known at compile time. We say that a type `T` is tuple-like for the parameter pack of types `Types` if it fulfills the following requirements.

## Requirements

1. The tuple `T` has a size accessible using template `std::tuple_size` whose type is equal to `std::size_t` and whose value is equal to `sizeof...(Types)`.
2. The type of each stored value in the tuple `T` is accessible using `std::tuple_element`, with the N'th stored value equal to the N'th type in `Types`.
3. Each stored value in `T` is accessible using the customization point object `grb::get`, which will invoke either the method `get()` if it is present in the tuple type `T` or `std::get()`. The type of the return value for the N'th element must be convertible to the N'th element of `Types`.

### Concept

_TODO: this concept could refactored to be a bit more elegant.

```
template <typename T, std::size_t I, typename U = grb::any>
concept TupleElementGettable = requires(T tuple) {
                                  {grb::get<I>(tuple)} -> std::convertible_to<U>;
                               };
template <typename T, typename... Args>
concept TupleLike =
  requires {
    typename std::tuple_size<std::remove_cvref_t<T>>::type;
    requires
        std::same_as<std::remove_cvref_t<decltype(std::tuple_size_v<std::remove_cvref_t<T>>)>,
        std::size_t>;
  } &&
  sizeof...(Args) == std::tuple_size_v<std::remove_cvref_t<T>> &&
  []<std::size_t... I>(std::index_sequence<I...>) {
```

```
    return (TupleElementGettable<T, I, Args> && ...);
  }(std::make_index_sequence<std::tuple_size_v<std::remove_cvref_t<T>>>());
```

### Example

```
// The Concept `TupleLike<int, int, float>` constraints `T`
// to be a tuple-like type storing int, int, float.
template<TupleLike<int, int, float> T>
void print_tuple(T&& tuple) {
  auto&& [i, j, v] = tuple;
  printf("%d, %d, %f\n", i, j, v);
}
```

## Matrix Entry

Matrix entries represent entries in a GraphBLAS matrix, which include both a tuple-like index storing the row and column index of the stored scalar value, as well as the scalar value itself. We say that a type `Entry` is a valid matrix entry for the scalar type `T` and index type `I` if the following requirements are met.

### Requirements

1. `Entry` is a tuple-like type with a size of 2.
2. The first element stored in the tuple-like type `Entry` is a tuple-like type fulfilling `TupleLike<I, I>`, storing the row and column index of the matrix entry.
3. The second element stored in the tuple-like type `Entry` holds the matrix entry's scalar value, and is convertible to `T`.

#### Concept

```
template <typename Entry, typename T, typename I>
concept MatrixEntry = TupleLike<Entry, grb::any, grb::any> &&
                      requires(Entry entry) { {grb::get<0>(entry)} -> TupleLike<I, I>; }
        &&
                      requires(Entry entry) { {grb::get<1>(entry)} ->
        std::convertible_to<T>; };
```

## Mutable Matrix Entry

A mutable matrix entry is an entry in a matrix that fulfills all the requirements of matrix entry, but whose stored scalar value can be mutated by assigning to some value of type `U`. We say that a matrix entry `Entry` is a mutable matrix entry for scalar type `T`, index type `I`, and output type `U`, if it fulfills all the requirements of matrix entry as well as the following requirements.

### Requirements

1. The second element of the tuple `Entry`, representing the scalar value, is assignable to elements of type `U`.

#### Concept

```
template <typename Entry, typename T, typename I, typename U>
concept MutableMatrixEntry = MatrixEntry<Entry, T, I> &&
                             std::is_assignable_v<decltype(std::get<1>
        (std::declval<Entry>())), U>;
```

## Matrix Range

A matrix in GraphBLAS consists of a range of values distributed over a two-dimensional domain. In addition to `grb::matrix`, which directly stores a collection of values, there are other types, such as views, that fulfill the same interface. We say that a type `M` is a matrix range if the following requirements are met.

### Requirements

1. `M` has a scalar type of the stored values, accessible with `grb::matrix_scalar_t<M>`
2. `M` has an index type used to reference the indices of the stored values, accessible with `grb::matrix_index_t<M>`.
3. `M` is a range with a value type that represents a matrix tuple, containing both the index and scalar value for each stored value.
4. `M` has a shape, which is a tuple-like object of size two, holding the number of rows and the number of columns, accessible by invoking the method `shape()` on an object of type `M`.
5. `M` has a method `find()` that takes an index tuple and returns an iterator.

#### Concept

```
template <typename M>
concept MatrixRange = std::ranges::sized_range<M> &&
  requires(M matrix) {
    typename grb::matrix_scalar_t<M>;
    typename grb::matrix_index_t<M>;
    {std::declval<std::ranges::range_value_t<std::remove_cvref_t<M>>>()}
      -> MatrixEntry<grb::matrix_scalar_t<M>,
                     grb::matrix_index_t<M>>;
    {grb::shape(matrix)} -> Tuplelike<grb::matrix_index_t<M>,
                                      grb::matrix_index_t<M>>;
    {grb::find(matrix, {grb::matrix_index_t<M>{}, grb::matrix_index_t<M>{}})}
                -> std::convertible_to<std::ranges::iterator_t<M>>;
  };
```

# Mutable Matrix Range

Some matrices and matrix-like objects are *mutable*, meaning that their stored values may be modified. Examples of mutable matrix ranges include instantiations of `grb::matrix` and certain matrix views that allow adding new values and modifying old values, such as `grb::transpose`. We say that a type `M` is a mutable matrix range for the scalar value `T` if the following requirements are met.

## Requirements

1. `M` is a matrix range.
2. The value type of `M` fulfills the requirements of `MutableMatrixEntry<T, I>`.
3. `M` has a method `insert()` that takes a matrix entry tuple and attempts to insert the element into the matrix, returning an iterator to the new element on success and returning an iterator to the end on failure.

### Concept

```
template <typename M, typename T>
concept MutableMatrixRange = MatrixRange<M> &&
                             MutableMatrixEntry<std::ranges::range_value_t<M>
                                                grb::matrix_scalar_t<M>,
                                                grb::matrix_index_t<M>,
                                                T> &&
  requires(M matrix, T value) {
    {grb::insert(matrix, {{grb::matrix_index_t<M>{}, grb::matrix_index_t<M>{}},
                          value}) -> std::ranges::iterator_t<M>;
    }
  };
```

# Mask Matrix Range

Some operations require masks, which can be used to avoid computing and storing certain parts of the output. We say that a type `M` is a mask matrix range if the following requirements are met.

## Requirements

1. `M` is a matrix range.
2. The scalar value type of `M` is convertible to `bool`.

### Concept

```
template <typename M>
concept MaskMatrixRange = MatrixRange<M> &&
                          std::is_convertible_v<grb::matrix_scalar_t<M>, bool>;
```

# Vector Entry

Vector entries represent entries in a GraphBLAS vector, which include both an `std::integral` index storing the index of the stored scalar value, as well as the scalar value itself. We say that a type `Entry` is a valid matrix entry for the scalar type `T` and index type `I` if the following requirements are met.

## Requirements

1. `Entry` is a tuple-like type with a size of 2.
2. The first element stored in the tuple-like type `Entry` fulfills `std::integral`.
3. The second element stored in the tuple-like type `Entry` holds the vector's scalar value, and is convertible to `T`.

### Concept

```
template <typename Entry, typename T, typename I>
concept VectorEntry = TupleLike<Entry, grb::any, grb::any> &&
                      requires(Entry entry) { {grb::get<0>(entry)} -> std::integral; }
        &&
                      requires(Entry entry) { {grb::get<1>(entry)} ->
        std::convertible_to<T>; };
```

# Mutable Vector Entry

A mutable vector entry is an entry in a vector that fulfills all the requirements of vector entry, but whose stored scalar value can be mutated by assigning to some value of type `U`. We say that a vector entry `Entry` is a mutable vector entry for scalar type `T`, index type `I`, and output type `U`, if it fulfills all the requirements of vector entry as well as the following requirements.

## Requirements

1. The second element of the tuple `Entry`, representing the scalar value, is assignable to elements of type `U`.

### Concept

```
template <typename Entry, typename T, typename I, typename U>
concept MutableVectorEntry = VectorEntry<Entry, T, I> &&
                             std::is_assignable_v<decltype(std::get<1>
        (std::declval<Entry>())), U>;
```

# Vector Range

A vector in GraphBLAS consists of a range of values distributed over a one-dimensional domain. In addition to `grb::vector`, which directly stores a collection of values, there are other types, such as views, that fulfill the same interface. We say that a type `V` is a vector range if the following requirements are met.

*TODO: make requirements list find CPO, not method.*

## Requirements

1. `V` has a scalar type of the stored values, accessible with `grb::vector_scalar_t<V>`
2. `V` has an index type used to reference the indices of the stored values, accessible with `grb::vector_index_t<V>`.
3. `V` is a range with a value type that represents a vector tuple, containing both the index and scalar value for each stored value.
4. `V` has a shape, which is an integer-like object, holding the dimension of the vector, accessible by invoking the method `shape()` on an object of type `V`.
5. `V` has a method `find()` that takes an index and returns an iterator.

### Concept

*TODO: this is a bit sketchy, and needs to have some of the components fleshed out.*

```
template <typename M>
concept VectorRange = std::ranges::sized_range<V> &&
  requires(V vector) {
    typename grb::vector_scalar_t<V>;
    typename grb::vector_index_t<V>;
    {std::declval<std::ranges::range_value_t<std::remove_cvref_t<V>>>()}
      -> VectorEntry<grb::vector_scalar_t<V>,
                     grb::vector_index_t<V>>;
    {grb::shape(vector)} -> std::same_as<grb::vector_index_t<V>>;
    {grb::find(vector, grb::vector_index_t<V>)} ->
        std::convertible_to<std::ranges::iterator_t<V>>;
  };
```

# Mutable Vector Range

Some vectors and vector-like objects are *mutable*, meaning that their stored values may be modified. Examples of mutable vector ranges include instantiations of `grb::vector` and certain vector views that allow adding new values and modifying old values, such as `grb::transpose`. We say that a type `V` is a mutable vector range for the scalar value `T` if the following requirements are met.

### Requirements

1. `V` is a vector range.
2. The value type of `M` fulfills the requirements of `MutableVectorEntry<T, I>`.
3. `M` has a method `insert()` that takes a vector entry tuple and attempts to insert the element into the vector returning an iterator to the new element on success and returning an iterator to the end on failure.

#### Concept

```
template <typename V, typename T>
concept MutableVectorRange = VectorRange<V> &&
                             MutableVectorEntry<std::ranges::range_value_t<V>,
                                                grb::vector_scalar_t<V>,
                                                grb::vector_index_t<V>,
                                                T> &&
  requires(V vector, T value) {
    {grb::insert(vector, {grb::vector_index_t<V>{}, value})}
      -> std::same_as<std::ranges::iterator_t<V>>;
  };
```

## Mask Vector Range

Some operations require masks, which can be used to avoid computing and storing certain parts of the output. We say that a type `M` is a mask vector range if the following requirements are met.

### Requirements

1. `M` is a vector range.
2. The scalar value type of `M` is convertible to `bool`.

#### Concept

```
template <typename M>
concept MaskVectorRange = VectorRange<M> &&
                          std::is_convertible_v<grb::vector_scalar_t<M>, bool>;
```

# Type Traits

## grb::monoid_traits

```
template <typename Fn, typename T>
struct monoid_traits;
```

The `monoid_traits` template struct provides information about the monoid that is formed by the commutative binary operator `Fn` on the type `T`. Namely, it provides a way to retrieve the monoid's identity.

Users can specialize `monoid_traits` for custom operators, which will make their identity elements available to GraphBLAS algorithms, possibly enabling optimizations. The default specialization of `monoid_traits` detects and provides the identity of the monoid if it has an `identity` method.

# Template Parameters

`Fn` - type of commutative binary operator

`T` - type on which `Fn` forms a commutative monoid

### Member Types

| Member Type | Definition | Description |
|---|---|---|
| `type` | `T` | type of the monoid |

The default specialization has the following member functions:

### Member Functions

| Function | Description |
|---|---|
| `identity` | the identity value of the monoid |

## grb::monoid_traits<Fn, T>::identity

```
static constexpr T identity() noexcept;
```

Returns the identity of the monoid.

#### Parameters

None

#### Complexity

Constant

### Exceptions

May not throw exceptions

### Possible Implementation

```
template <typename Fn, typename T>
static constexpr T grb::monoid_traits<Fn, T>::identity() noexcept {
  if constexpr(requires { {Fn::identity() } -> std::same_as<T> }) {
    return Fn::identity();
  } else if constexpr(requires { {Fn:: template identity<T>()} -> std::same_as<T> }) {
    return Fn:: template identity<T>();
  }
}
```

## Specializations

```
template <std::arithmetic T>
struct grb::monoid_traits<std::plus<T>, T>;

template <std::arithmetic T>
struct grb::monoid_traits<std::plus<void>, T>;

template <std::arithmetic T>
struct grb::monoid_traits<std::multiplies<T>, T>;

template <std::arithmetic T>
struct grb::monoid_traits<std::multiplies<void>, T>;
```

### grb::matrix_value_t

```
template <typename Matrix>
using matrix_value_t = std::ranges::range_value_t<Matrix>;
```

Obtain the value type of the matrix-like type `Matrix`. Equal to `std::ranges::range_value_t<Matrix>`.

### grb::vector_value_t

```
template <typename Vector>
using vector_value_t = std::ranges::range_value_t<Vector>;
```

Obtain the value type of the vector-like type `Vector`. Equal to `std::ranges::range_value_t<Vector>`.

### grb::matrix_scalar_t

```
template <typename Matrix>
using matrix_scalar_t = std::remove_cvref_t<typename std::tuple_element<1,
        matrix_value_t<Matrix>>::type>;
```

Obtain the type of scalar values stored in the matrix-like type `Matrix`. Equal to the second element stored in the matrix's tuple-like value type.

### grb::vector_scalar_t

```
template <typename Vector>
using vector_scalar_t = std::remove_cvref_t<typename std::tuple_element<1,
        vector_value_t<Vector>>::type>;
```

Obtain the type of scalar values stored in the vector-like type `Vector`. Equal to the second element stored in the vector's tuple-like value type.

### grb::matrix_key_t

```
template <typename Matrix>
using matrix_key_t = std::remove_cvref_t<typename std::tuple_element<0,
        matrix_value_t<Matrix>>::type>;
```

Obtain the key type of the matrix. This is a tuple-like type used to store each scalar value's row and column indices.

### grb::matrix_index_t

```
template <typename Matrix>
using matrix_index_t = std::remove_cvref_t<typename std::tuple_element<0,
        matrix_key_t<Matrix>>::type>;
```

Obtain the integer-like type used to store matrix indices.

# Algorithms

## Execution Policy

Execution policies are objects that indicate the kind of parallelism allowed when executing an algorithm. For versions of GraphBLAS algorithms that support execution policies, users may pass in the execution policies defined in the C++ standard library.

- `std::execution::sequenced_policy`
- `std::execution::parallel_policy`
- `std::execution::parallel_unsequenced_policy`

- `std::execution::unsequenced_policy`

GraphBLAS implementations may also provide implementation-defined execution policies. Implementations must define the execution behavior for any execution policies they define.

# Exceptions and Error Handling

Exceptions may be thrown to indicate error conditions. A number of exceptions are defined which will be thrown in different error conditions.

## GraphBLAS-Defined Exceptions

- `grb::out_of_range` is thrown if an index is provided which lies outside the dimensions of the matrix, or if a value is not present at the index.
- `grb::matrix_io::file_error` is thrown if a file cannot be read.
- `grb::invalid_argument` is thrown if objects' dimensions are incompatible.

## Other Exceptions

- Some functions and methods may allocate temporary memory. These may throw

`std::bad_alloc`. - Some functions and methods may allocate memory using a user-provided allocator, which may throw.

- Some functions and methods accept user-defined binary or unary operators. These may throw.

# Multiply

The function `grb::multiply` in GraphBLAS is used to multiply a matrix times a matrix, a matrix times a vector, or a vector times a matrix, depending on the types of the input arguments.

### Matrix Times Matrix

```
template <MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>, grb::matrix_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
          MaskMatrixRange M = grb::full_matrix_mask<>
>
multiply_result_t<A, B, Reduce, Combine>
multiply(A&& a,
         (1)
```

```
          B&& b,
          Reduce&& reduce = Reduce{},
          Combine&& combine = Combine{},
          M&& mask = M{});

template <MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>, grb::matrix_scalar_t<B>> Combine =
        grb::multiplies<>,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce = grb::plus<>,
          MaskMatrixRange M = grb::full_matrix_mask<>,
          MutableMatrixRange<grb::combine_result_t<A, B, Combine>> C,
          BinaryOperator<grb::matrix_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
>
void multiply(C&& c,
        (2)
              A&& a,
              B&& b,
              Reduce&& reduce = Reduce{},
              Combine&& combine = Combine{},
              M&& mask = M{},
              Accumulate&& acc = Accumulate{},
              bool merge = false);

template <typename ExecutionPolicy,
          MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>, grb::matrix_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
          MaskMatrixRange M = grb::full_matrix_mask<>
>
multiply_result_t<A, B, Reduce, Combine>
multiply(ExecutionPolicy&& policy,
        (3)
          A&& a,
          B&& b,
          Reduce&& reduce = Reduce{},
          Combine&& combine = Combine{},
          M&& mask = M{});

template <typename ExecutionPolicy,
```

```
          MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>, grb::matrix_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
          MaskMatrixRange M = grb::full_matrix_mask<>,
          BinaryOperator<grb::matrix_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableMatrixRange<grb::combine_result_t<A, B, Combine>> C
>
void multiply(ExecutionPolicy&& policy,
        (4)
              C&& c,
              A&& a,
              B&& b,
              Reduce&& reduce = Reduce{},
              Combine&& combine = Combine{},
              M&& mask = M{},
              Accumulate&& acc = Accumulate{},
              bool merge = false);
```

### Matrix Times Vector

```
template <MatrixRange A,
          VectorRange B,
          BinaryOperator<grb::matrix_scalar_t<A>, grb::vector_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
          MaskVectorRange M = grb::full_vector_mask<>
>
mutiply_result<A, B, Reduce, Combine>
multiply(A&& a,
        (5)
         B&& b,
         Reduce&& reduce = Reduce{},
         Combine&& combine = Combine{},
         M&& mask = M{});



template <MatrixRange A,
          VectorRange B,
          BinaryOperator<grb::matrix_scalar_t<A>, grb::vector_scalar_t<B>> Combine =
        grb::multiplies<>,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
```

```
                        grb::combine_result_t<A, B, Combine>,
                        grb::combine_result_t<A, B, Combine>> Reduce = grb::plus<>,
          MaskVectorRange M = grb::full_vector_mask<>,
          MutableVectorRange<grb::combine_result_t<A, B, Combine>> C,
          BinaryOperator<grb::vector_scalar_t<C>,
                        grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right<>
>
void multiply(C&& c,
        (6)
              A&& a,
              B&& b,
              Reduce&& reduce = Reduce{},
              Combine&& combine = Combine{},
              M&& mask = M{},
              Accumulate&& acc = Accumulate{},
              bool merge = false);

template <typename ExecutionPolicy,
          MatrixRange A,
          VectorRange B,
          BinaryOperator<grb::matrix_scalar_t<A>, grb::vector_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                        grb::combine_result_t<A, B, Combine>,
                        grb::combine_result_t<A, B, Combine>> Reduce,
          MaskVectorRange M = grb::full_vector_mask<>
>
mutiply_result<A, B, Reduce, Combine>
multiply(A&& a,
        (7)
         B&& b,
         Reduce&& reduce = Reduce{},
         Combine&& combine = Combine{},
         M&& mask = M{});

template <typename ExecutionPolicy,
          MatrixRange A,
          VectorRange B,
          BinaryOperator<grb::matrix_scalar_t<A>, grb::vector_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                        grb::combine_result_t<A, B, Combine>,
                        grb::combine_result_t<A, B, Combine>> Reduce,
          MaskVectorRange M = grb::full_vector_mask<>,
          BinaryOperator<grb::matrix_scalar_t<C>,
                        grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableVectorRange<grb::combine_result_t<A, B, Combine>> C
```

```
>
void multiply(C&& c,
        (8)
              A&& a,
              B&& b,
              Reduce&& reduce = Reduce{},
              Combine&& combine = Combine{},
              M&& mask = M{},
              Accumulate&& acc = Accumulate{},
              bool merge = false);
```

### Vector Times Vector

```
template <VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t<A>, grb::vector_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
>
mutiply_result<A, B, Reduce, Combine>
multiply(A&& a,
         (9)
         B&& b,
         Reduce&& reduce = Reduce{},
         Combine&& combine = Combine{});

template <VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t<A>, grb::vector_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
          BinaryOperator<grb::vector_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          std::assignable_from<grb::combine_result_t<A, B, Combine>> C
>
void multiply(C&& c,
        (10)
              A&& a,
              B&& b,
              Reduce&& reduce = Reduce{},
              Combine&& combine = Combine{},
              Accumulate&& acc = Accumulate{},
              bool merge = false);
```

```
template <typename ExecutionPolicy,
          VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t<A>, grb::vector_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
>
mutiply_result<A, B, Reduce, Combine>
multiply(ExecutionPolicy&& policy,
         (11)
         A&& a,
         B&& b,
         Reduce&& reduce = Reduce{},
         Combine&& combine = Combine{});

template <typename ExecutionPolicy,
          VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t<A>, grb::vector_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
          BinaryOperator<grb::vector_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          std::assignable_from<grb::combine_result_t<A, B, Combine>> C
>
void multiply(ExecutionPolicy&& policy,
         (12)
              C&& c,
              A&& a,
              B&& b,
              Reduce&& reduce = Reduce{},
              Combine&& combine = Combine{},
              Accumulate&& acc = Accumulate{},
              bool merge = false);
```

### Vector Times Matrix

```
template <VectorRange A,
          MatrixRange B,
          BinaryOperator<grb::vector_scalar_t<A>, grb::matrix_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
          MaskVectorRange M = grb::full_vector_mask<>
```

```
>
mutiply_result<A, B, Reduce, Combine>
multiply(A&& a,
         (13)
         B&& b,
         Reduce&& reduce = Reduce{},
         Combine&& combine = Combine{},
         M&& mask = M{});


template <VectorRange A,
          MatrixRange B,
          BinaryOperator<grb::vector_scalar_t<A>, grb::matrix_scalar_t<B>> Combine =
        grb::multiplies<>,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce = grb::plus<>,
          MaskVectorRange M = grb::full_vector_mask<>,
          MutableVectorRange<grb::combine_result_t<A, B, Combine>> C,
          BinaryOperator<grb::vector_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
>
void multiply(C&& c,
         (14)
              A&& a,
              B&& b,
              Reduce&& reduce = Reduce{},
              Combine&& combine = Combine{},
              M&& mask = M{},
              Accumulate&& acc = Accumulate{},
              bool merge = false);


template <typename ExecutionPolicy,
          VectorRange A,
          MatrixRange B,
          BinaryOperator<grb::vector_scalar_t<A>, grb::matrix_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
          MaskVectorRange M = grb::full_vector_mask<>
>
mutiply_result<A, B, Reduce, Combine>
multiply(ExecutionPolicy&& policy,
         (15)
         A&& a,
         B&& b,
         Reduce&& reduce = Reduce{},
         Combine&& combine = Combine{},
```

```
          M&& mask = M{});

template <typename ExecutionPolicy,
          VectorRange A,
          MatrixRange B,
          BinaryOperator<grb::vector_scalar_t<A>, grb::matrix_scalar_t<B>> Combine,
          BinaryOperator<grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>,
                         grb::combine_result_t<A, B, Combine>> Reduce,
          MaskVectorRange M = grb::full_vector_mask<>,
          BinaryOperator<grb::vector_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableVectorRange<grb::combine_result_t<A, B, Combine>> C
>
void multiply(ExecutionPolicy&& policy,
         (16)
               C&& c,
               A&& a,
               B&& b,
               Reduce&& reduce = Reduce{},
               Combine&& combine = Combine{},
               M&& mask = M{},
               Accumulate&& acc = Accumulate{},
               bool merge = false);
```

Behavior is non-deterministic if `reduce` is not associative or not commutative.

## Parameters

`policy` - the execution policy to use

`a` - the left-hand side of the product being computed

`b` - the right-hand side of the product being computed

`reduce` - a binary operator

`combine` - a binary operator

`mask` - write mask used to determine which elements of the output will be computed

`c` - if given, the output object to which to write the result

`acc` - the accumulator used to accumulate partial results into the output object

`merge` - whether to merge in values from `c` outside of the write area indicated by `mask`

**Type Requirements**

- `A` must meet the requirements of `MatrixRange` (1-8) or `VectorRange` (9-16)
- `B` must meet the requirements of `MatrixRange` (1-4,13-16) or `VectorRange` (5-12)
- `Reduce` must meet the requirements of `BinaryOperator<grb::combine_result_type_t<A, B, Combine>, grb::combine_result_type_t<A, B, Combine>, grb::combine_result_type_t<A, B, Combine>>`
- `Combine` must meet the requirements of `BinaryOperator<grb::container_value_t<A>, grb::container_value_t<B>>`
- `M` must meet the requirements of `MaskMatrixRange` (1-8) or `VectorMaskRange` (9-16)

## Return Value

If the output matrix or vector argument, `c`, is supplied, no value is returned.

**NOTE: `combine_result_t<A, B, Combine> is actually incorrect here. Should be result of reduction.**

If `c` is not supplied as an argument, returns the result of the multiplication.

1. ◦ A GraphBLAS matrix with shape `a.shape()[0]` by `b.shape()[1]`
2. ◦ A GraphBLAS vector with shape `a.shape()[0]`
3. ◦ `combine_result_t<A, B, Combine>`
4. ◦ A GraphBLAS vector with shape `b.shape()[1]`

In the case that a GraphBLAS matrix or vector is returned, its scalar type is `combine_result_t<A, B, Combine>`, and its index type is equal to either `grb::matrix_index_t<A>` or `grb::matrix_index_t<B>`, whichever one has the larger `std::numeric_limits<T>::max()`. An element of the result at index location `index` will only be computed if a scalar value equal to `true` when converted to `bool` exists in `mask`, that is `grb::find(mask, index) != grb::end(mask) && bool(grb::get<1>(*grb::find(mask, index)))`.

### *Matrix Times Matrix*

A generalized matrix times matrix multiplication is performed, as defined in the GraphBLAS Math Specification. Each element of the output is produced by combing the elements in corresponding indices of a row of `a` and column of `b` and using `reduce` to perform a reduction to a single value.

### *Matrix Times Vector*

A generalized matrix times vector multiplication is performed, as defined in the GraphBLAS Math Specification. Each element of the output vector is produced by combining elements in each row of `a` with the corresponding elements of the vector `b` and using `reduce` to perform a reduction of these to a single value in the output vector.

#### *Vector Times Vector*

A generalized dot product is performed, as defined in the GraphBLAS Math Specification. The corresponding elements of vectors `a` and `b` are combined and reduced into a single value using `reduce`.

#### *Vector Times Matrix*

A generalized matrix times vector multiplication is performed, as defined in the GraphBLAS Math Specification. Each element of the output vector is produced by combining elements in each column of `b` with the corresponding elements of the vector `a` and using `reduce` to perform a reduction of these to a single value in the output vector.

## Complexity

Complexity is implementation-defined.

## Exceptions

The exception `grb::invalid_argument` may be thrown if any of the following conditions occur:

#### *Matrix Times Matrix*

- The dimensions of the matrices being multiplied are incompatible, that is `a.shape()[1] != b.shape()[0]`.
- The dimensions of the mask are smaller than the dimensions of the output, that is `mask.shape()[0] < a.shape()[0] || mask.shape()[1] < b.shape()[1]`
- The dimensions of the output matrix, if provided, are incompatible, that is `c.shape()[0] != a.shape()[0] || c.shape()[1] != b.shape()[1]`.

#### *Matrix Times Vector*

- The dimensions of the matrix and vector being multiplied are incompatible, that is `a.shape()[1] != b.shape()`.
- The dimensions of the mask are smaller than the dimensions of the output, that is `mask.shape() < a.shape()[0]`
- The dimensions of the output vector, if provided, are incompatible, that is `c.shape() != a.shape()[0]`.

#### *Vector Times Vector*

- The dimensions of the vectors being multiplied are incompatible, that is `a.shape() != b.shape()`.
- The dimensions of the mask are smaller than the dimensions of the output, that is `mask.shape() < a.shape()[0]`
- The dimensions of the output vector, if provided, are incompatible, that is `c.shape() != a.shape()[0]`.

***Vector Times Matrix***

- The dimensions of the matrix and vector being multiplied are incompatible, that is `a.shape() != b.shape()[0]`.
- The dimensions of the mask are smaller than the dimensions of the output, that is `mask.shape() < b.shape()[1]`
- The dimensions of the output vector, if provided, are incompatible, that is `c.shape() != b.shape()[1]`.

If the algorithm fails to allocate memory, `std::bad_alloc` is thrown.

## Notes

## Example

# ewise_union

Perform an element-wise union between two GraphBLAS matrices or vectors.

### Element-Wise Matrix Union

```
template <MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>,
                         grb::matrix_scalar_t<B>> Combine,
          MaskMatrixRange M = grb::full_matrix_mask<>
>
ewise_union_result_t<A, B, Combine>
ewise_union(A&& a, B&& b, Combine&& combine, M&& mask = M{});                (1)

template <MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>,
                         grb::matrix_scalar_t<B>> Combine,
          MaskMatrixRange M = grb::full_matrix_mask<>,
          BinaryOperator<grb::matrix_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableMatrixRange<grb::combine_result_t<A, B, Combine>> C
>
void ewise_union(C&& c, A&& a, B&& b,
                 Combine&& combine, M&& mask = M{},
                 Accumulate&& acc = Accumulate{},
                 bool merge = false);                                        (2)
```

```
template <typename ExecutionPolicy,
          MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>,
                         grb::matrix_scalar_t<B>> Combine,
          MaskMatrixRange M = grb::full_matrix_mask<>
>
ewise_union_result_t<A, B, Combine>
ewise_union(ExecutionPolicy&& policy,
            A&& a, B&& b,
            Combine&& combine, M&& mask = M{});                              (3)

template <typename ExecutionPolicy,
          MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>,
                         grb::matrix_scalar_t<B>> Combine,
          MaskMatrixRange M = grb::full_matrix_mask<>,
          BinaryOperator<grb::matrix_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableMatrixRange<grb::combine_result_t<A, B, Combine>> C
>
void ewise_union(ExecutionPolicy&& policy,
                 C&& c, A&& a, B&& b,
                 Combine&& combine, M&& mask = M{},
                 Accumulate&& acc = Accumulate{},
                 bool merge = false);                                        (4)
```

### Element-Wise Vector Union

```
template <VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t<A>,
                         grb::vector_scalar_t<B>> Combine,
          MaskVectorRange M = grb::full_vector_mask<>
>
ewise_union_result_t<A, B, Combine>
ewise_union(A&& a, B&& b, Combine&& combine, M&& mask = M{});                (5)

template <VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t_t<A>,
                         grb::vector_scalar_t_t<B>> Combine,
          MaskVectorRange M = grb::full_vector_mask<>,
```

```
          BinaryOperator<grb::matrix_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableVectorRange<grb::combine_result_t<A, B, Combine>> C
>
void ewise_union(C&& c, A&& a, B&& b,
                 Combine&& combine, M&& mask = M{},
                 Accumulate&& acc = Accumulate{},
                 bool merge = false);                                    (6)


template <typename ExecutionPolicy,
          VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t<A>,
                         grb::vector_scalar_t<B>> Combine,
          MaskVectorRange M = grb::full_vector_mask<>
>
ewise_union_result_t<A, B, Combine>
ewise_union(ExecutionPolicy&& policy,
            A&& a, B&& b,
            Combine&& combine, M&& mask = M{});                          (7)

template <typename ExecutionPolicy,
          VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t_t<A>,
                         grb::vector_scalar_t_t<B>> Combine,
          MaskVectorRange M = grb::full_vector_mask<>,
          BinaryOperator<grb::matrix_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableVectorRange<grb::combine_result_t<A, B, Combine>> C
>
void ewise_union(ExecutionPolicy&& policy,
                 C&& c, A&& a, B&& b,
                 Combine&& combine, M&& mask = M{},
                 Accumulate&& acc = Accumulate{},
                 bool merge = false);                                    (8)
```

## Parameters

`policy` - the execution policy to use

`a` - matrix or vector on the left-hand side of the element-wise operation

`b` - matrix or vector on the right-hand side of the element-wise operation

`c` - output matrix or vector in which to store the result of the multiply operation

`combine` - binary operator used to combine elements of `a` and `b`

`mask` - determines which parts of the output matrix will be computed

`acc` - binary operator used to combine elements of the result with stored elements in corresponding locations in `c`

`merge` - flag declaring whether to merge in elements of `c` outside the range indicated by `mask`

**Type Requirements**

- `A` must meet the requirements of `MatrixRange` (1,2,3,4) or `VectorRange` (5,6,7,8).
- `B` must meet the requirements of `MatrixRange` (1,2,3,4) or `VectorRange` (5,6,7,8).
- `C` must meet the requirements of `MutableMatrixRange<grb::combine_result_t<A, B, Combine>>` (2,4) or `MutableVectorRange<grb::combine_result_t<A, B, Combine>>` (6,8).
- `Combine` must meet the requirements of `BinaryOperator<grb::matrix_scalar_t<A>, grb::matrix_scalar_t<B>>` (1,2,3,4) or `BinaryOperator<grb::vector_scalar_t<A>, grb::vector_scalar_t<B>>` (5,6,7,8).
- `M` must meet the requirements of `MaskMatrixRange` (1,2,3,4) or `MaskVectorRange` (5,6,7,8).
- `Accumulate` must meet the requirements of `BinaryOperator<grb::matrix_scalar_t<C>, grb::combine_result_t<A, B, Combine>>` (2,4) or `BinaryOperator<grb::vector_scalar_t<C>, grb::combine_result_t<A, B, Combine>>` (6,8).

## Return Value

If the output matrix or vector parameter, `c`, is supplied, no value is returned.

If the parameter `c` is not supplied, the function returns a GraphBLAS matrix (1,3) or GraphBLAS vector (5,7) equal to the element-wise union of `a` and `b`, with the binary operator `combine` used to combine scalar values stored at the same index and `mask` used to determine which parts of the output are computed. For (1,3), the type of the return value satisfies the requirements of `MatrixRange`, and for (5,7) the type of the return value satisfies the requirements of `VectorRange`. The return value has the same shape as `a` and `b`. Index `idx` in the return value holds a stored value if and only if an element exists at that index in both `a` or `b` and an element equal to `true` when cast to `bool` exists at that index in `mask`. If a value exists at `idx` in `a` but not in `b`, the return value will hold a value equal to `a[idx]`. If a value exists in `b` but not in `a`, it will hold a value equal to `b[idx]`. If a value exists at `idx` in both `a` and `b`, it will hold a value equal to `fn(a[idx], b[idx])`.

## Preconditions

The parameters `a` and `b` must share the same shape. If an output object `c` is given, it must also have the same shape. For the parameter `mask`, each dimension of its shape must be equal to or greater than the corresponding dimension of `a` and `b`'s shapes. `fn` must not modify any element of `a`, `b`, or `mask`.

## Postconditions

In (2,4) and (6,8), an element-wise union is performed as described in the GraphBLAS Math Specification and the result written to `c`. In (1,3) and (5,7), none of the input arguments will be modified, and the result is returned as a value.

## Exceptions

The exception `grb::invalid_argument` may be thrown if any of the following conditions occur:

- The dimensions of `a` and `b` incompatible, that is `a.shape() != b.shape()`.
- The dimensions of `c`, if given, does not match `a`'s shape, that is `c.shape() != a.shape()`
- The dimensions of the mask are smaller than the dimensions of the output, that is `mask.shape()[0] < a.shape()[0] || mask.shape()[1] < a.shape()[1]` (1,2,3,4) or `mask.shape() < a.shape()` in (5,6,7,8).

## Notes

## Example

```
#include <grb/grb.hpp>

int main(int argc, char** argv) {
  grb::matrix<float> x({10, 10});
  grb::matrix<float> y({10, 10});

  x[{0, 1}] = 12;
  x[{2, 5}] = 12;
  x[{2, 7}] = 12;
  x[{5, 3}] = 12;

  y[{0, 1}] = 12;
  y[{1, 5}] = 12;
  y[{2, 7}] = 12;
  y[{5, 3}] = 12;

  auto z = grb::ewise_union(x, y, grb::plus{});

  grb::print(z);
```

```
  return 0;
}
```

## ewise_intersection

Perform an element-wise intersection between two GraphBLAS matrices or vectors.

### Element-Wise Matrix Intersection

```
template <MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>,
                         grb::matrix_scalar_t<B>> Combine,
          MaskMatrixRange M = grb::full_matrix_mask<>
>
ewise_intersection_result_t<A, B, Combine>
ewise_intersection(A&& a, B&& b, Combine&& combine, M&& mask = M{});          (1)

template <MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>,
                         grb::matrix_scalar_t<B>> Combine,
          MaskMatrixRange M = grb::full_matrix_mask<>,
          BinaryOperator<grb::matrix_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableMatrixRange<grb::combine_result_t<A, B, Combine>> C
>
void ewise_intersection(C&& c, A&& a, B&& b,
                        Combine&& combine, M&& mask = M{},
                        Accumulate&& acc = Accumulate{},
                        bool merge = false);                                  (2)


template <typename ExecutionPolicy,
          MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>,
                         grb::matrix_scalar_t<B>> Combine,
          MaskMatrixRange M = grb::full_matrix_mask<>
>
ewise_intersection_result_t<A, B, Combine>
ewise_intersection(ExecutionPolicy&& policy,
                   A&& a, B&& b,
```

```
                    Combine&& combine, M&& mask = M{});                          (3)

template <typename ExecutionPolicy,
          MatrixRange A,
          MatrixRange B,
          BinaryOperator<grb::matrix_scalar_t<A>,
                         grb::matrix_scalar_t<B>> Combine,
          MaskMatrixRange M = grb::full_matrix_mask<>,
          BinaryOperator<grb::matrix_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableMatrixRange<grb::combine_result_t<A, B, Combine>> C
>
void ewise_intersection(ExecutionPolicy&& policy,
                        C&& c, A&& a, B&& b,
                        Combine&& combine, M&& mask = M{},
                        Accumulate&& acc = Accumulate{},
                        bool merge = false);                                     (4)
```

### Element-Wise Vector Intersection

```
template <VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t_t<A>,
                         grb::vector_scalar_t_t<B>> Combine,
          MaskVectorRange M = grb::full_vector_mask<>
          >
ewise_intersection_result_t<A, B, Combine, M>
ewise_intersection(A&& a, B&& b, Combine&& combine, M&& mask = M{});             (5)

template <VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t_t<A>,
                         grb::vector_scalar_t_t<B>> Combine,
          MaskVectorRange M = grb::full_vector_mask<>,
          BinaryOperator<grb::matrix_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableVectorRange<grb::combine_result_t<A, B, Combine>> C
>
void ewise_intersection(C&& c, A&& a, B&& b,
                        Combine&& combine, M&& mask = M{},
                        Accumulate&& acc = Accumulate{},
                        bool merge = false);                                     (6)
```

```
template <typename ExecutionPolicy,
          VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t_t<A>,
                         grb::vector_scalar_t_t<B>> Combine,
          MaskVectorRange M = grb::full_vector_mask<>
          >
ewise_intersection_result_t<A, B, Combine, M>
ewise_intersection(ExecutionPolicy&& policy,
                   A&& a, B&& b,
                   Combine&& combine, M&& mask = M{});                    (7)

template <typename ExecutionPolicy,
          VectorRange A,
          VectorRange B,
          BinaryOperator<grb::vector_scalar_t_t<A>,
                         grb::vector_scalar_t_t<B>> Combine,
          MaskVectorRange M = grb::full_vector_mask<>,
          BinaryOperator<grb::matrix_scalar_t<C>,
                         grb::combine_result_t<A, B, Combine>> Accumulate =
        grb::take_right,
          MutableVectorRange<grb::combine_result_t<A, B, Combine>> C
>
void ewise_intersection(ExecutionPolicy&& policy,
                        C&& c, A&& a, B&& b,
                        Combine&& combine, M&& mask = M{},
                        Accumulate&& acc = Accumulate{},
                        bool merge = false);                              (8)
```

Perform an element-wise intersection.

## Parameters

`policy` - the execution policy to use

`a` - matrix or vector on the left-hand side of the element-wise operation

`b` - matrix or vector on the right-hand side of the element-wise operation

`c` - output matrix or vector in which to store the result of the multiply operation

`combine` - binary operator used to combine elements of `a` and `b`

`mask` - determines which parts of the output matrix will be computed

`acc` - binary operator used to combine elements of the result with stored elements in corresponding locations in `c`

`merge` - flag declaring whether to merge in elements of `c` outside the range indicated by `mask`

**Type Requirements**

- `A` must meet the requirements of `MatrixRange` (1,2,3,4) or `VectorRange` (5,6,7,8).
- `B` must meet the requirements of `MatrixRange` (1,2,3,4) or `VectorRange` (5,6,7,8).
- `C` must meet the requirements of `MutableMatrixRange<grb::combine_result_t<A, B, Combine>>` (2,4) or `MutableVectorRange<grb::combine_result_t<A, B, Combine>>` (6,8).
- `Combine` must meet the requirements of `BinaryOperator<grb::matrix_scalar_t<A>, grb::matrix_scalar_t<B>>` (1,2,3,4) or `BinaryOperator<grb::vector_scalar_t<A>, grb::vector_scalar_t<B>>` (5,6,7,8).
- `M` must meet the requirements of `MaskMatrixRange` (1,2,3,4) or `MaskVectorRange` (5,6,7,8).
- `Accumulate` must meet the requirements of `BinaryOperator<grb::matrix_scalar_t<C>, grb::combine_result_t<A, B, Combine>>` (2,4) or `BinaryOperator<grb::vector_scalar_t<C>, grb::combine_result_t<A, B, Combine>>` (6,8).

## Return Value

If the output matrix or vector parameter, `c`, is supplied, no value is returned.

If the parameter `c` is not supplied, the function returns a GraphBLAS matrix (1) or GraphBLAS vector (3) equal to the element-wise intersection of `a` and `b`, with the binary operator `combine` used to combine scalar values and `mask` used to determine which parts of the output are computed. For (1), the type of the return value satisfies the requirements of `MatrixRange`, and for (3) the type of the return value satisfies the requirements of `VectorRange`. The return value has the same shape as `a` and `b`. Index `idx` will only hold an element in the return value if an element exists at `idx` in both `a`, `b`, and `mask`, and the element of `mask` holds a value equal to `true` when converted to `bool`. The value at that index will be equal to the value `fn(a[idx], b[idx])`.

## Preconditions

The parameters `a` and `b` must share the same shape. If an output object `c` is given, it must also have the same shape. For the parameter `mask`, each dimension of its shape must be equal to or greater than the corresponding dimension of `a` and `b`'s shapes. `fn` must not modify any element of `a`, `b`, or `mask`.

## Postconditions

In (2,4) and (6,8), an element-wise intersection is performed as described in the GraphBLAS Math Specification and the result written to `c`. In (1,3) and (5,7), none of the input arguments will be modified, and the result is returned as a value.

### Exceptions

The exception `grb::invalid_argument` may be thrown if any of the following conditions occur:

- The dimensions of `a` and `b` incompatible, that is `a.shape() != b.shape()`.
- The dimensions of `c`, if given, does not match `a`'s shape, that is `c.shape() != a.shape()`
- The dimensions of the mask are smaller than the dimensions of the output, that is `mask.shape()[0] < a.shape()[0] || mask.shape()[1] < a.shape()[1]` (1,2,3,4) or `mask.shape() < a.shape()` (5,6,7,8).

### Notes

### Example

```
#include <grb/grb.hpp>

int main(int argc, char** argv) {
  grb::matrix<float> x({10, 10});
  grb::matrix<float> y({10, 10});

  x[{0, 1}] = 12;
  x[{2, 5}] = 12;
  x[{2, 7}] = 12;
  x[{5, 3}] = 12;

  y[{0, 1}] = 12;
  y[{1, 5}] = 12;
  y[{2, 7}] = 12;
  y[{5, 3}] = 12;

  auto z = grb::ewise_intersection(x, y, grb::multiplies{});

  grb::print(z);

  return 0;
}
```

## assign

```
template <grb::MatrixRange B,
          grb::MutableMatrixRange<grb::matrix_scalar_t<B>> A>
void assign(A&& a, B&& b);                                          (1)

template <typename T, grb::MutableMatrixRange A<T>>
```

```
void assign(A&& a, const T& value);                                    (2)


template <grb::VectorRange V,
          grb::MutableVectorRange<grb::vector_scalar_t<V> U>
void assign(U&& u, V&& v);                                             (3)


template <typename T, grb::MutableVectorRange V<T>>
void assign(U&& u, const T& value);                                    (4)
```

Assigns every element of a GraphbLAS matrix or vector to either (1, 3) another GraphBLAS matrix or vector with the same shape, or (2, 4) a single scalar value.

## Parameters

`a` - the matrix to be assigned

`b` - the matrix whose contents will be assigned to `a`

`u` - the vector to be assigned

`v` - the vector whose contents will be assigned to `v`

`value` - a scalar value to be assigned to every index location in `a`

### Type Requirements

- `A` must be a mutable matrix range for the scalar value being assigned, that is it must meet the requirements of `grb::MutableMatrixRange<grb::matrix_scalar_t<B>>` (1) or `grb::MutableMatrixRange<T>` (2)
- `B` must meet the requirements of `grb::MatrixRange`
- `U` must be a mutable vector range for the scalar value being assigned, that is it must meet the requirements of `grb::MutableVectorRange<grb::vector_scalar_t<V>` (3) or `grb::MutableVectorRange<T>` (4)
- `V` must meet the requirements of `grb::VectorRange`

## Exceptions

The exception `grb::invalid_argument` is thrown if `a` and `b` or `u` and `v` are not the same shape. `std::bad_alloc` may be thrown if `a` or `u`'s allocator is unable to allocate memory.

Other exceptions may be thrown while assigning or constructing to `a` or `u`.