GraphBLAS:

Building a C++ Matrix API for Graph Algorithms

BENJAMIN BROCK, SCOTT MCMILLAN

Cppcon
The C++ Conference
2021
October 24-29

About Us

Ben, PhD Candidate at **UC Berkeley**

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

Scott, Principal Engineer at **CMU SEI**

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

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

DM21-0916

This Talk

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

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

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

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

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

This Talk

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

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

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

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

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

This Talk

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

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

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

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

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

This Talk

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

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

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

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

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

This Talk

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

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

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

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

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

What This Talk Is Not

- A C++ **standards proposal**

- A complete **evaluation** of graph programming models

What This Talk Is **Not**

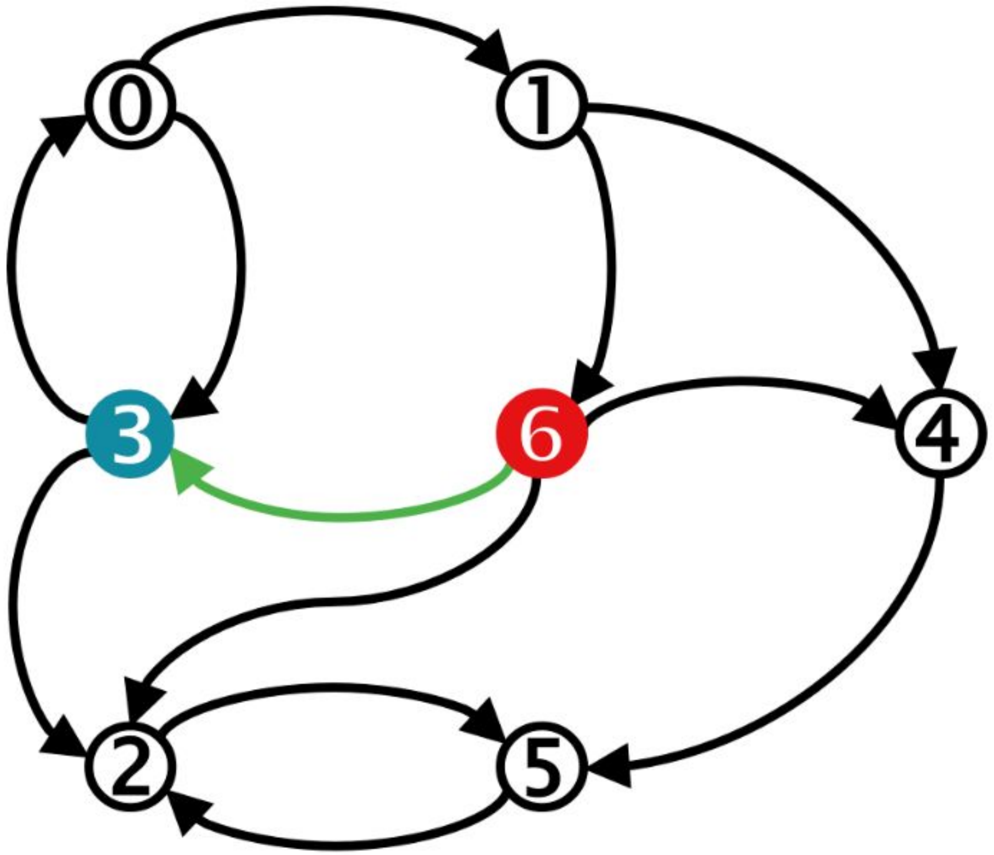
- A C++ **standards proposal**

- A complete **evaluation** of graph programming models

Background: How and why to use matrix algebra for graphs

Graphs: Understanding relationships between items

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



Graph is a pair (V, E):

- V is a set of vertices
- E is a set of paired vertices (edges)

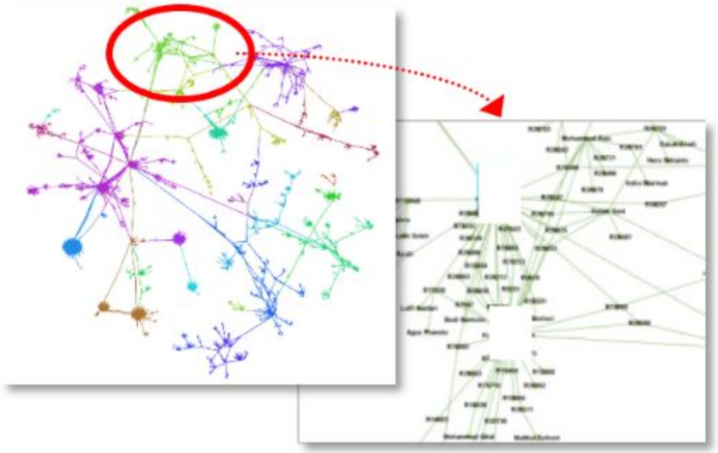
$V = \{0, 1, 2, 3, 4, 5, 6\}$

$E = \{(0,1), (0,3), (1,4), (1,6), (2,5), (3,0),$
$(3,2), (4,5), (5,2), (6,2), (6, 3), (6,4)\}$

Ordered pairs results in directed graphs (shown)

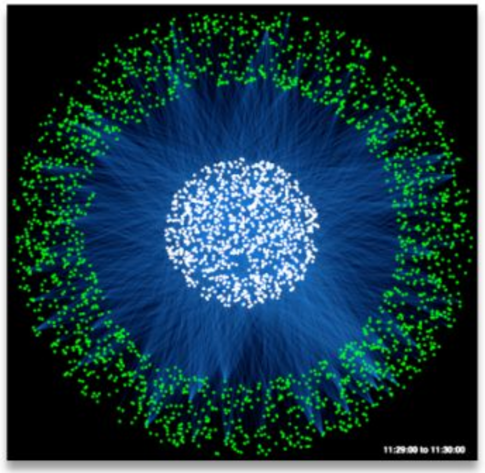
Graph Analysis is *Important* and *Pervasive*

Social



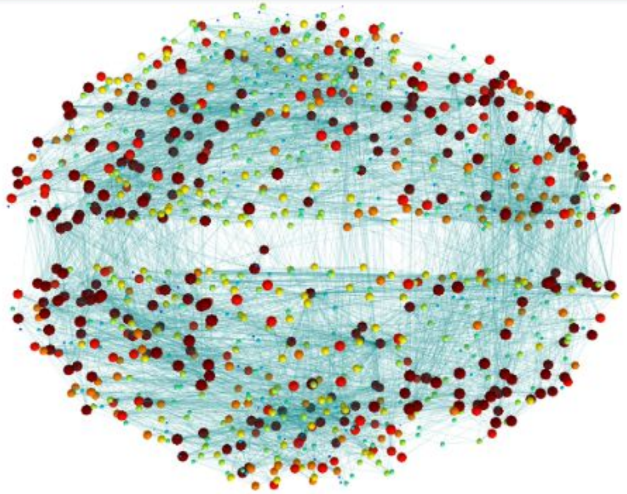
- Graphs represent relationships between individuals or documents
- 100,000s – 100,000,000s individuals and interactions

Cyber



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

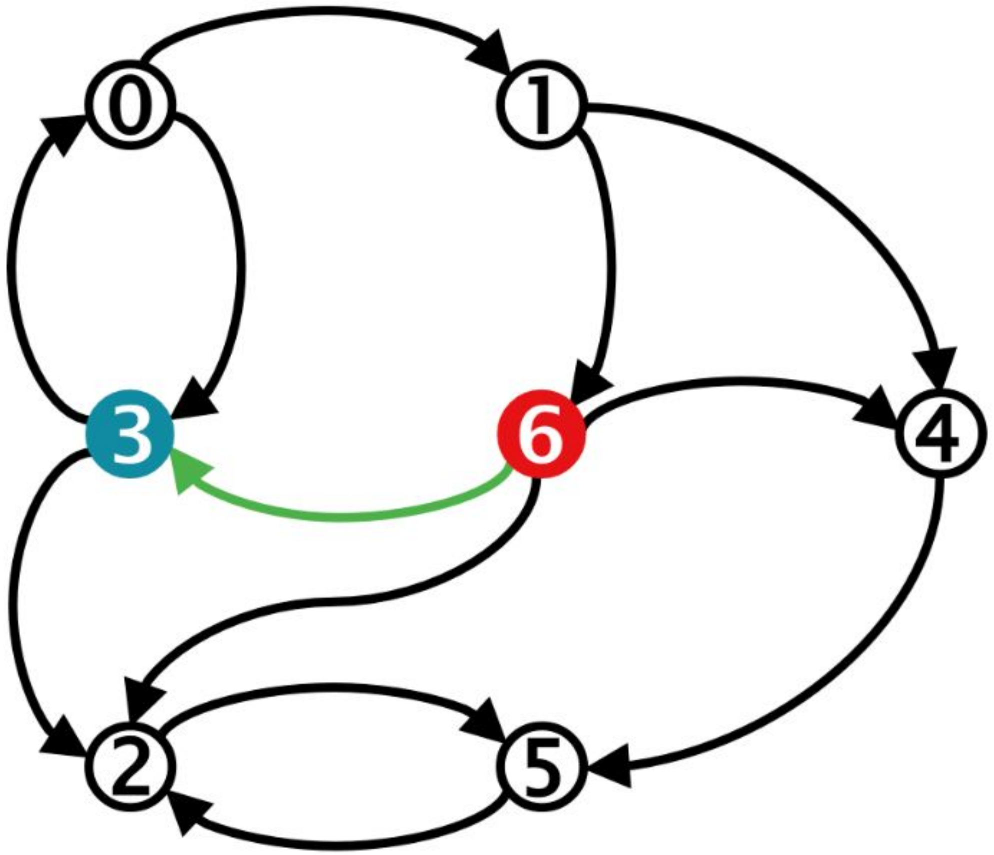
Biology



- Graphs represent organization of neural interactions within the brain
- 10^{11} – 10^{15} neurons and connections

Graphs as Adjacency Matrices

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



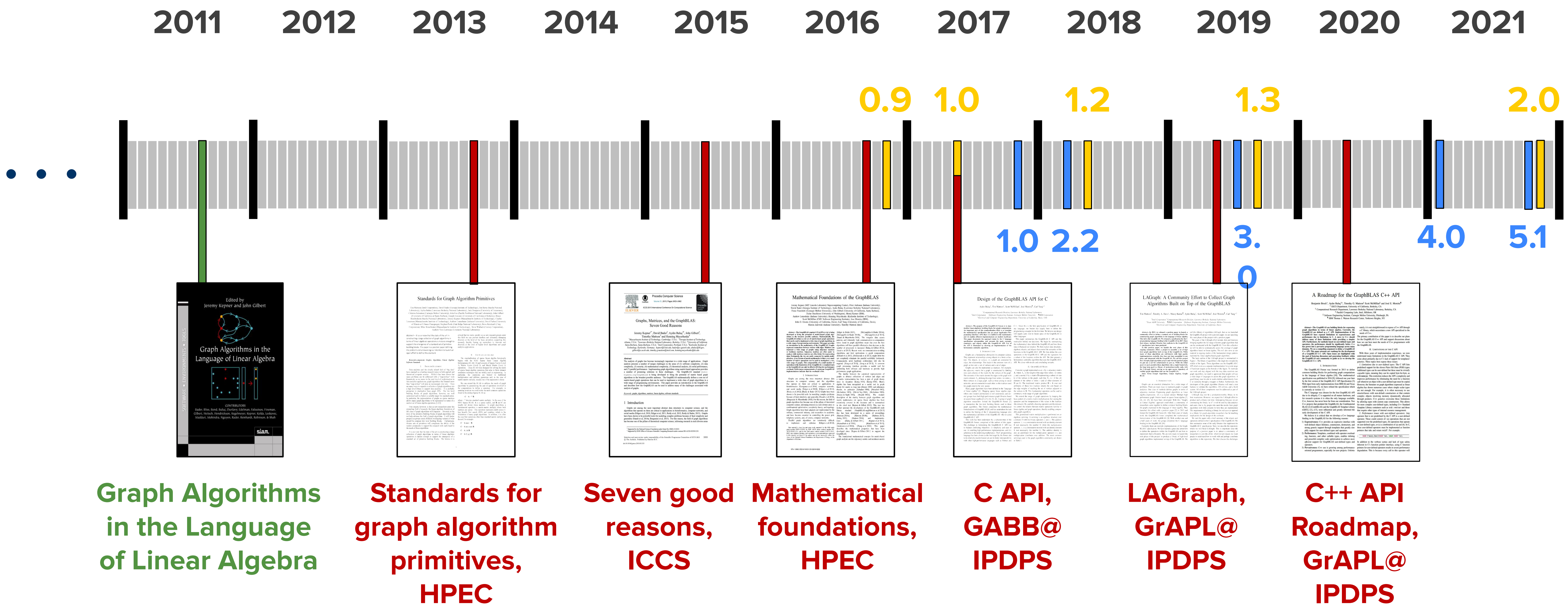
dest.

A	0	1	2	3	4	5	6
0		●		●			
1					●		●
2						●	
3	●		●				
4						●	
5			●				
source 6			●	●	●		

$$\mathbf{A}_{ij} = \begin{cases} \bullet & (v_i, v_j) \in E \\ \emptyset & (v_i, v_j) \notin E \end{cases}$$

GraphBLAS Timeline

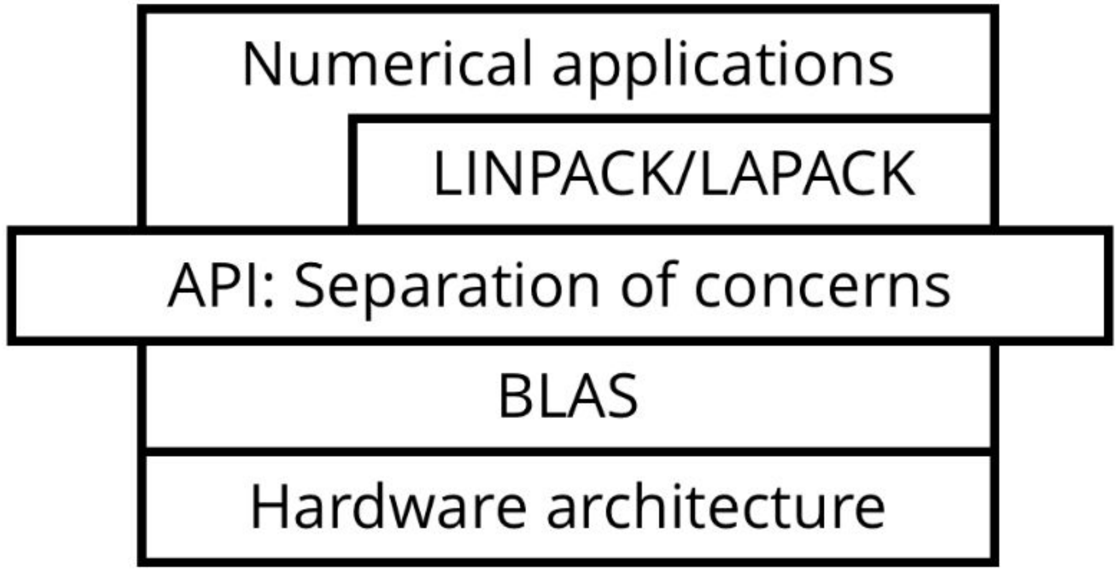
Book — Papers — GraphBLAS standards — SuiteSparse:GraphBLAS releases



The GraphBLAS “standard”

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

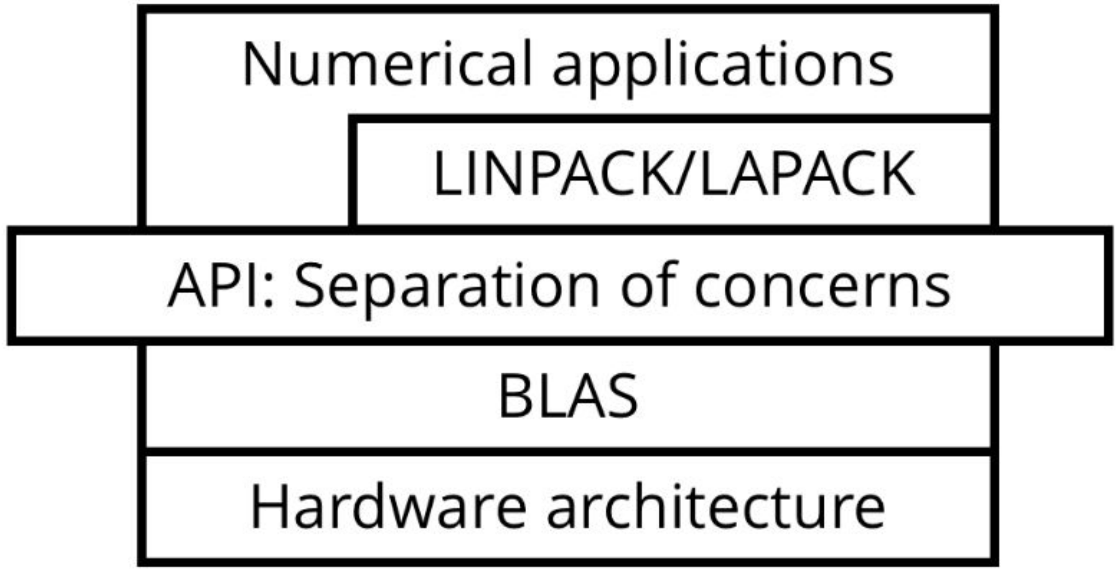
1979: BLAS Basic Linear Algebra Subprograms (BLAS 2 ’88, BLAS 3 ’90)



The GraphBLAS “standard”

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

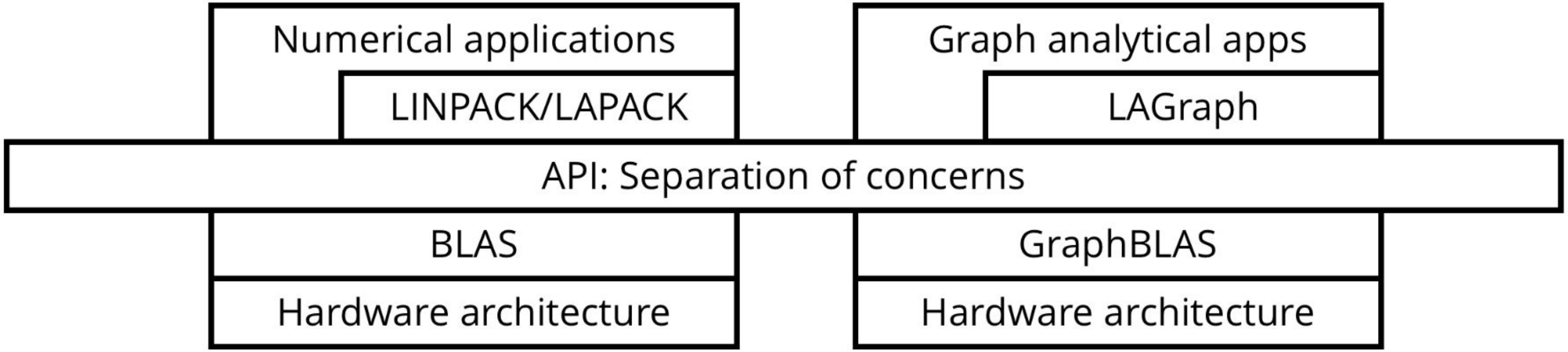
1979: BLAS	Basic Linear Algebra Subprograms (BLAS 2 ’88, BLAS 3 ’90)
2001: Sparse BLAS	an extension to BLAS (little uptake)



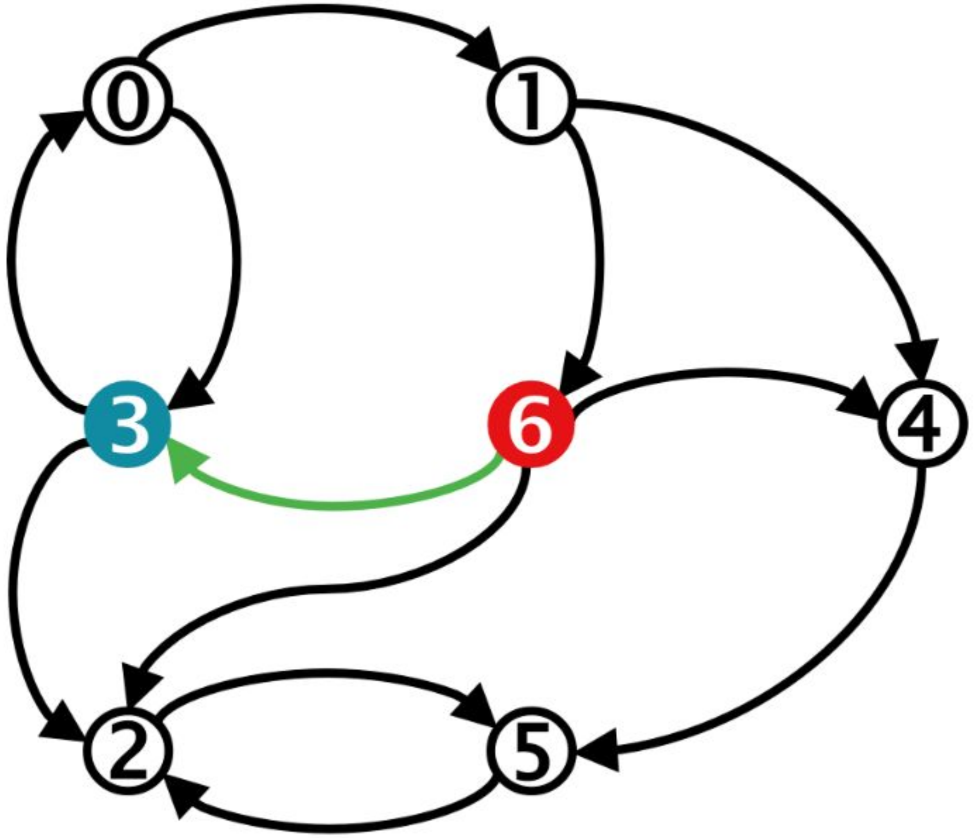
The GraphBLAS "standard"

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

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



Graphs as Adjacency Matrices

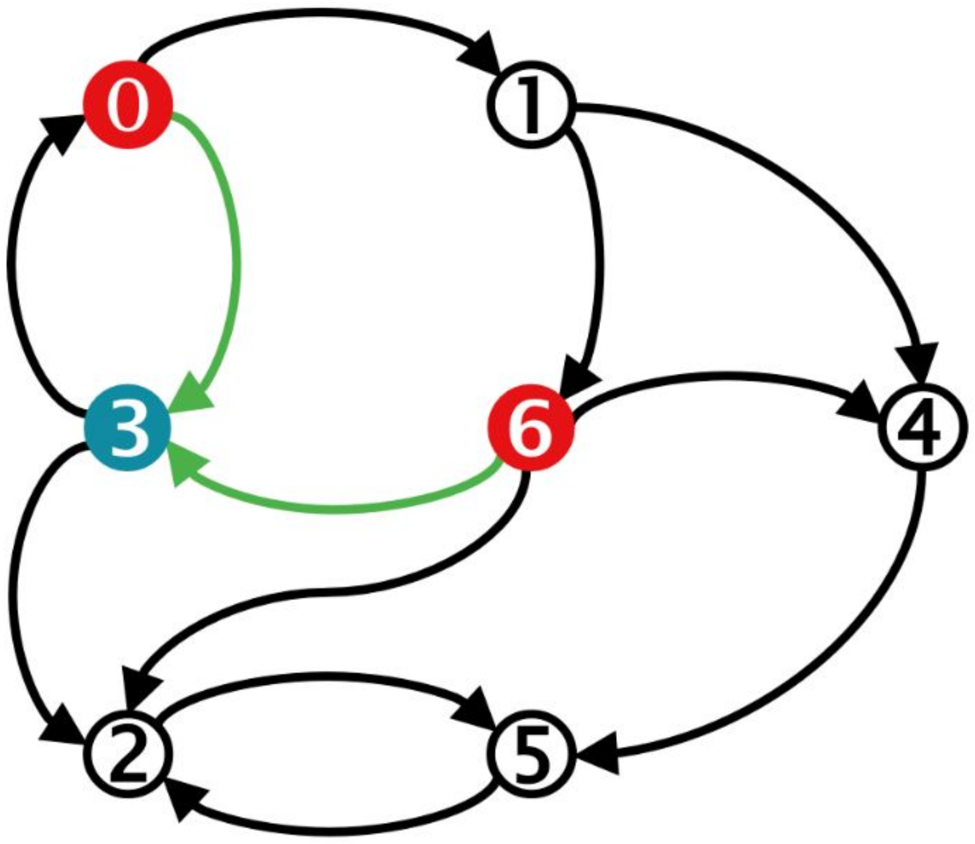


dest.

A	0	1	2	3	4	5	6
0		●		●			
1					●		●
2						●	
3	●		●				
4						●	
5			●				
source 6			●	●	●		

$$\mathbf{A}_{ij} = \begin{cases} \bullet & (v_i, v_j) \in E \\ \varnothing & (v_i, v_j) \notin E \end{cases}$$

Graphs as Adjacency Matrices

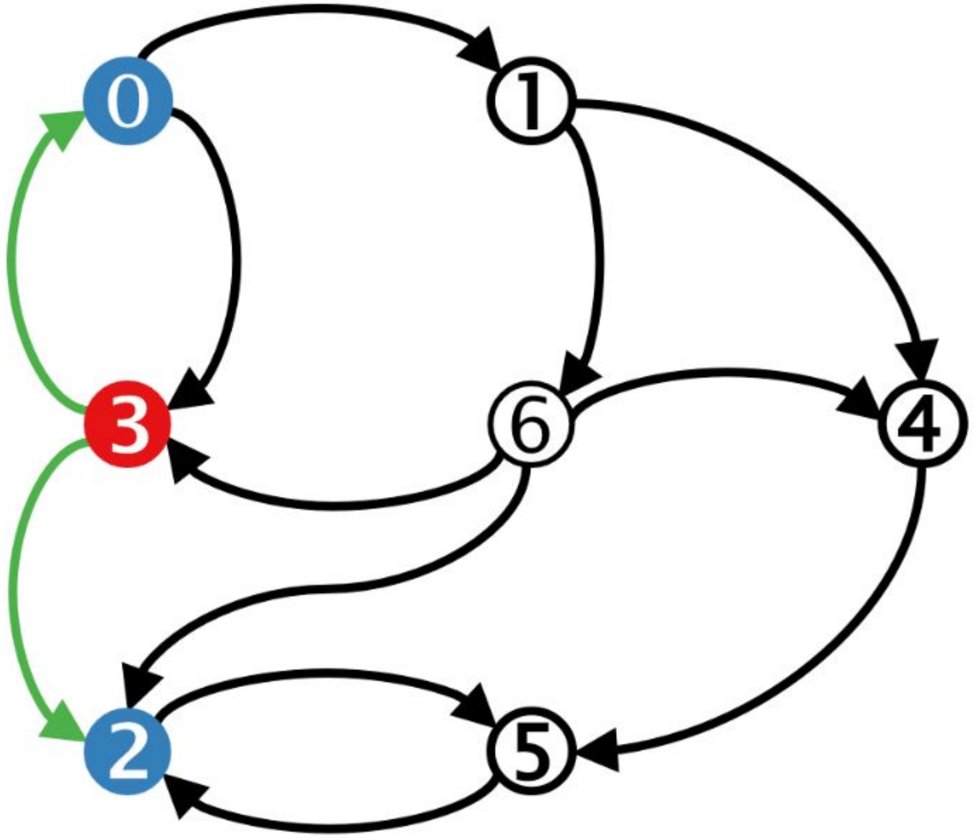


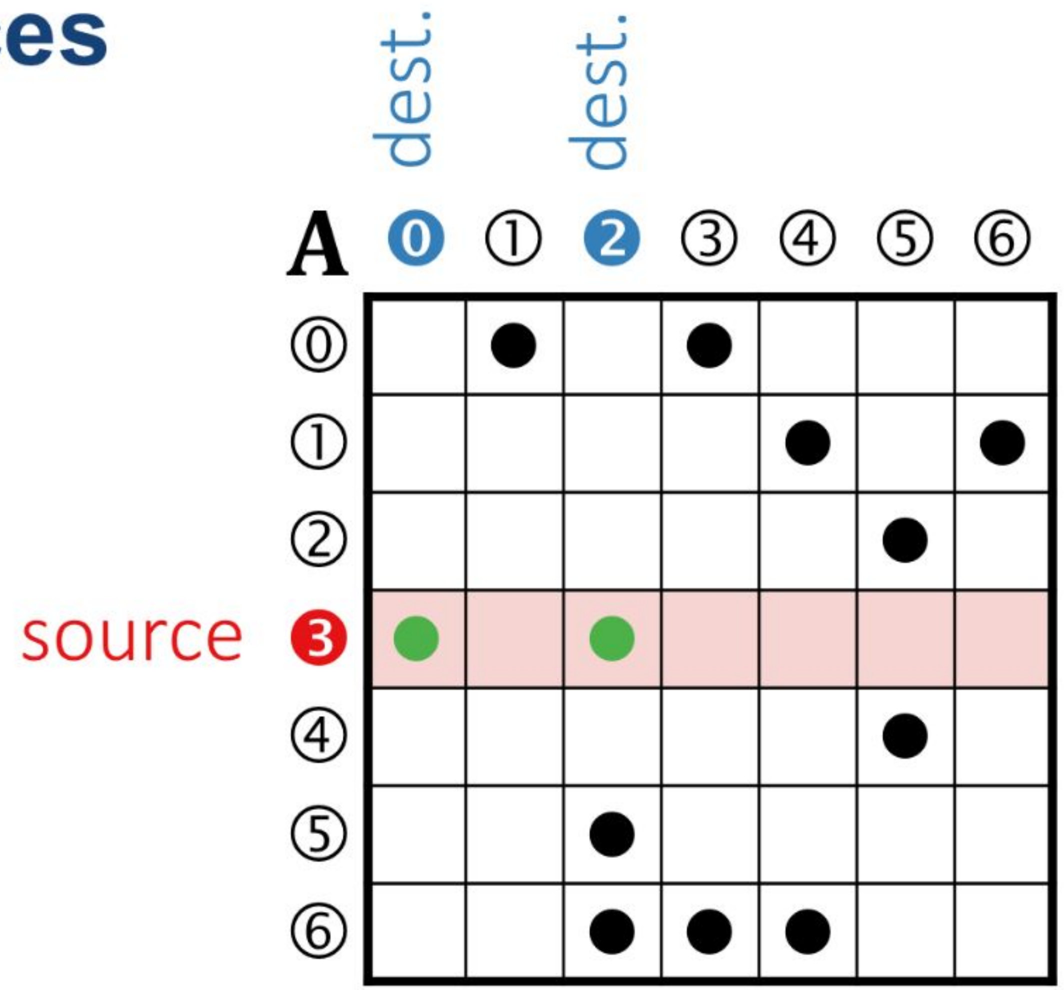
dest.

A	0	1	2	3	4	5	6
source 0		●		●			
1					●		●
2						●	
3	●		●				
4						●	
5			●				
source 6			●	●	●		

$$\mathbf{A}_{ij} = \begin{cases} \bullet & (v_i, v_j) \in E \\ \emptyset & (v_i, v_j) \notin E \end{cases}$$

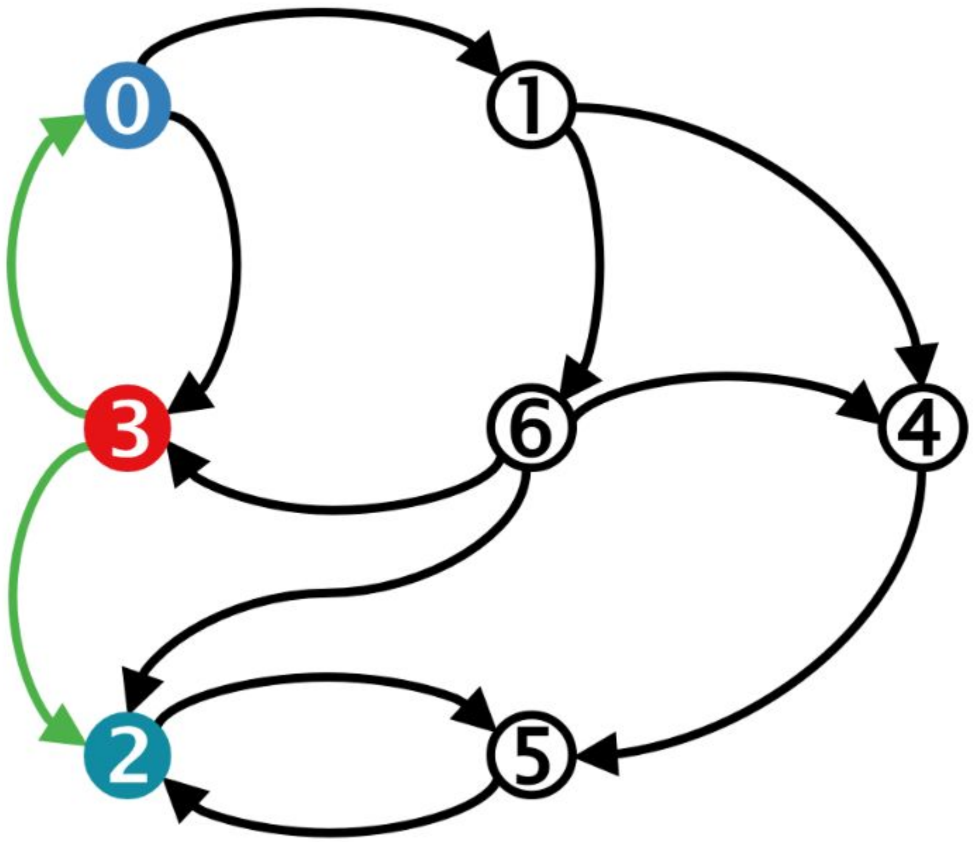
Graphs as Adjacency Matrices

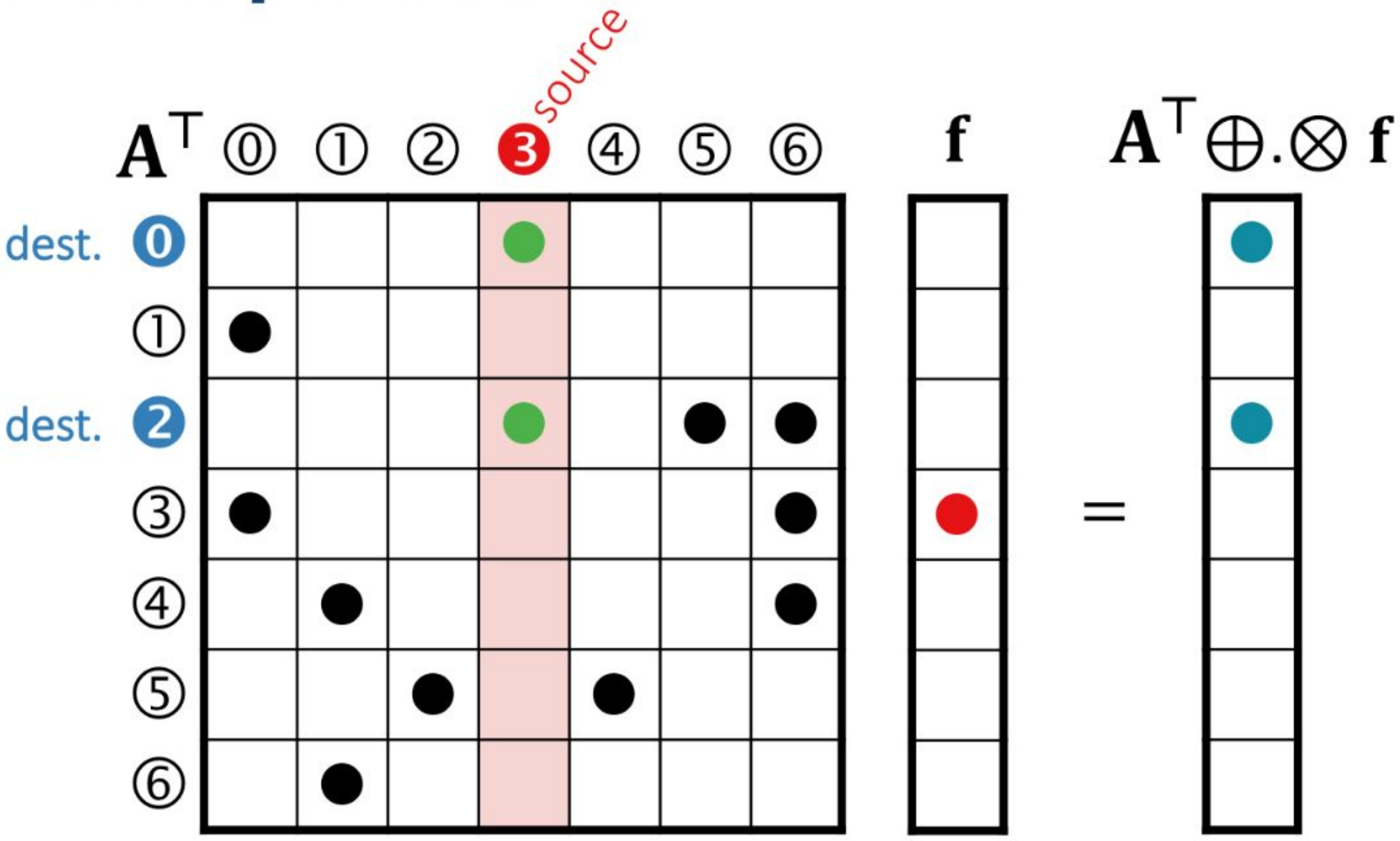




$$\mathbf{A}_{ij} = \begin{cases} \bullet & (v_i, v_j) \in E \\ \text{Ø} & (v_i, v_j) \notin E \end{cases}$$

Graph Operations as Matrix Operations

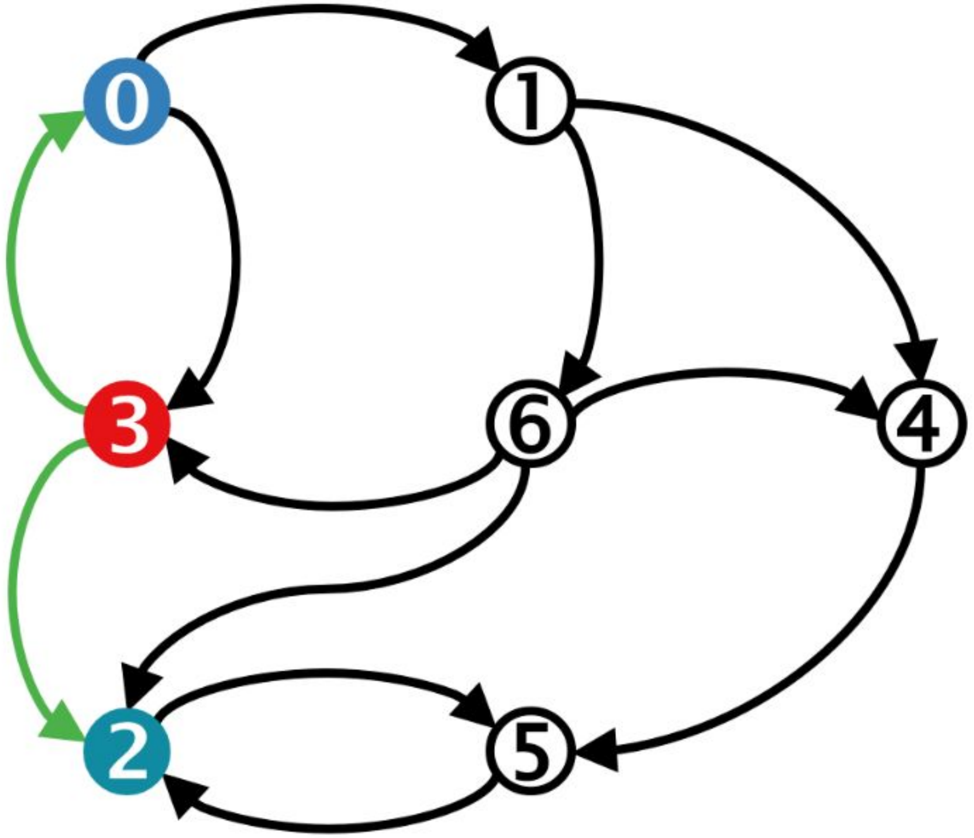


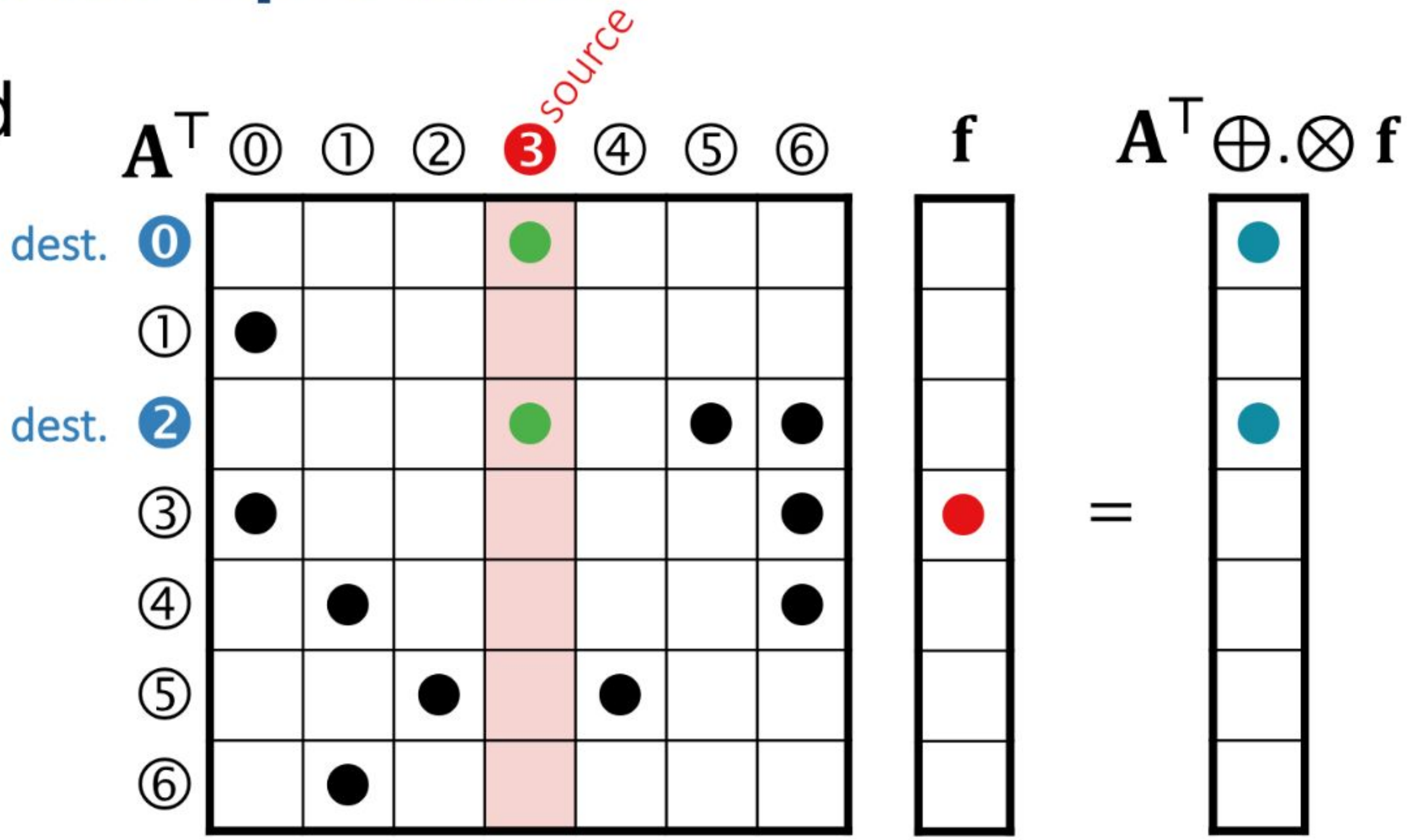


- Matrix-vector multiply → find neighbors
  - In-neighbors: use A
  - Out-neighbors: use A^T

Graph Operations as Matrix Operations

Finding out-neighbors is used many graph algorithms.

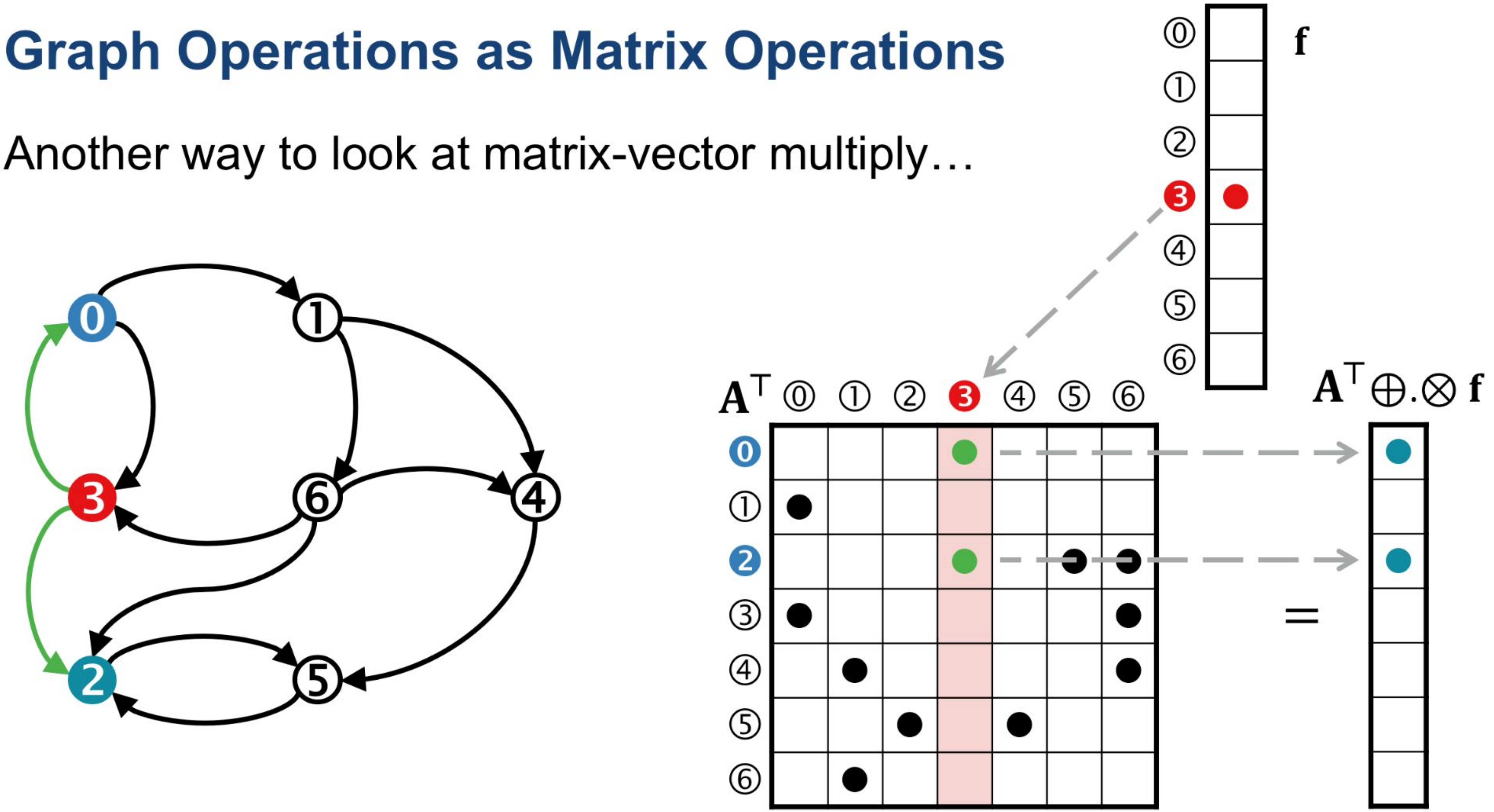




- Matrix-vector multiply → find neighbors
  - In-neighbors: use A
  - Out-neighbors: use A^T

Graph Operations as Matrix Operations

Another way to look at matrix-vector multiply…



Carnegie Mellon University
Software Engineering Institute
Berkeley
UNIVERSITY OF CALIFORNIA

What is $\oplus . \otimes$??

Matrix multiplication

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

$$\mathbf{y} = \mathbf{A}\,\mathbf{x}$$

$$\mathbf{y}(i) = \sum_{k} \mathbf{A}(i,k) \cdot \mathbf{x}(k)$$

Matrix multiplication on semirings

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

$$\mathbf{y} = \mathbf{A}\,\mathbf{x}$$

$$\mathbf{y}(i) = \sum_k \mathbf{A}(i,k) \cdot \mathbf{x}(k)$$

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

$$\mathbf{y} = \mathbf{A} \oplus.\otimes\ \mathbf{x}$$

$$\mathbf{y}(i) = \bigoplus_k \mathbf{A}(i,k) \otimes \mathbf{x}(k)$$

Matrix multiplication on semirings

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

$$\mathbf{y} = \mathbf{A}\,\mathbf{x}$$

$$\mathbf{y}(i) = \sum_{k} \mathbf{A}(i,k) \cdot \mathbf{x}(k)$$

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

$$\mathbf{y} = \mathbf{A} \oplus . \otimes \, \mathbf{x}$$

$$\mathbf{y}(i) = \bigoplus_{k} \mathbf{A}(i,k) \otimes \mathbf{x}(k)$$

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

GraphBLAS semirings ⊕.⊗

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

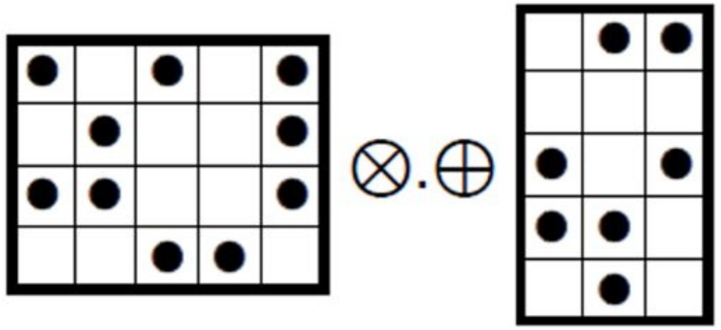
Semiring	Valid values	$\oplus$	$\otimes$	$\mathbf{0}$	Graph semantics
integer arithmetic	$a \in \mathbb{N}$	$+$	$\cdot$	0	number of paths
real arithmetic	$a \in \mathbb{R}$	$+$	$\cdot$	0	strength of all paths
boolean	$a \in \{\text{false}, \text{true}\}$	$\vee$	$\wedge$	false	connectivity
min-plus (tropical)	$a \in \mathbb{R} \cup \{+\infty\}$	min	$+$	$+\infty$	shortest path
max-plus	$a \in \mathbb{R} \cup \{-\infty\}$	max	$+$	$-\infty$	longest path

*In GraphBLAS this is not enforced nor required

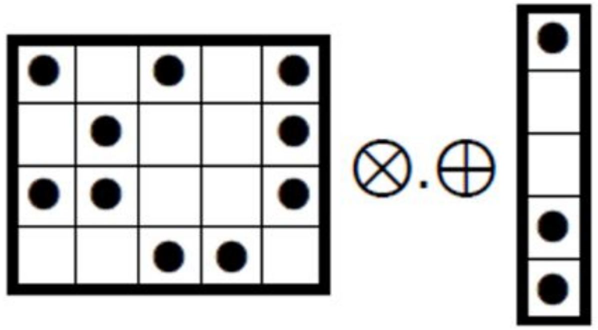
GraphBLAS Primitives

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

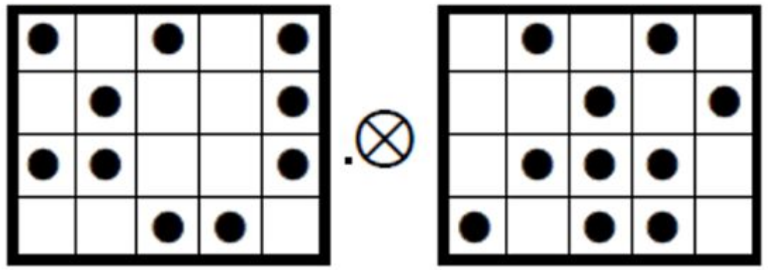
Sparse matrix times sparse matrix



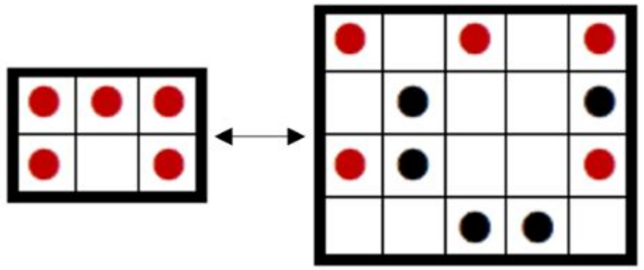
Sparse matrix times sparse vector



Element-wise multiplication (and addition)



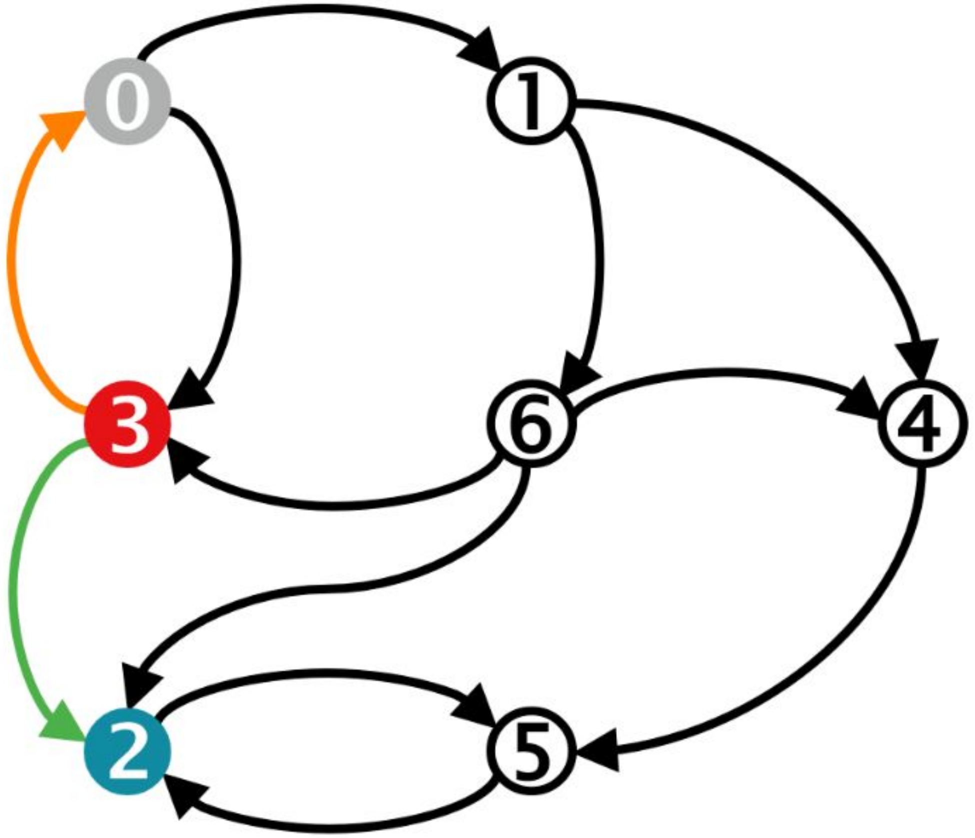
Sparse matrix extraction (and assignment)

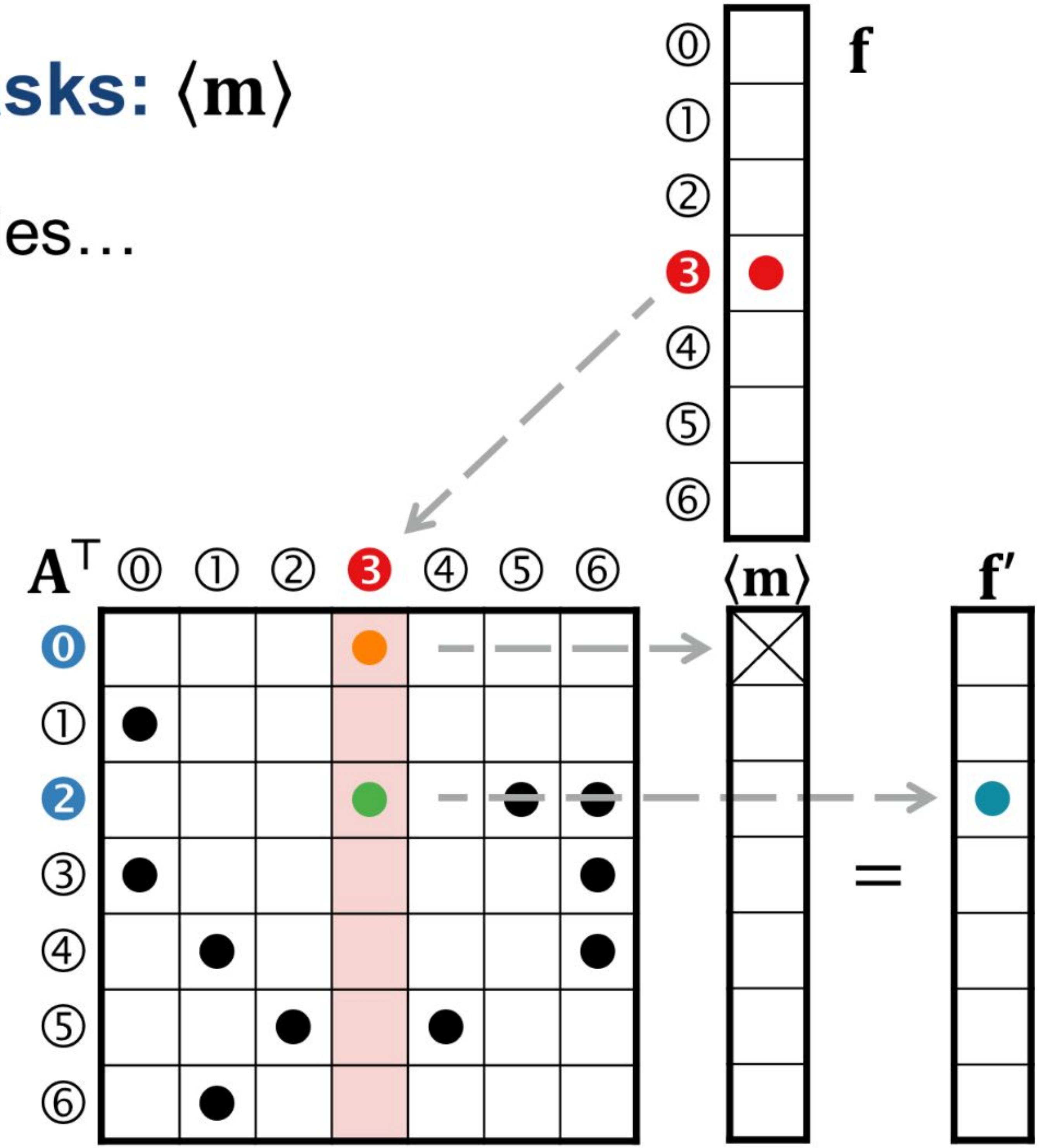


…plus reduction, transpose, Kronecker product, filtering, transform, etc.

One more thing… write masks: ⟨m⟩

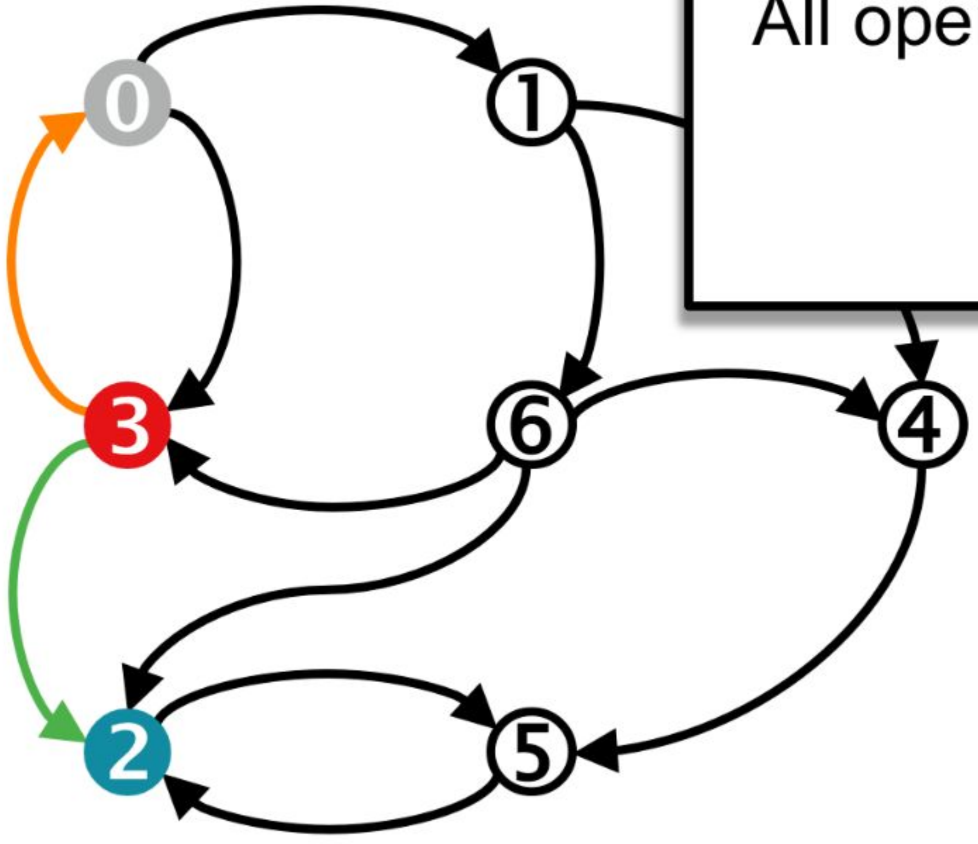
Often not interested in some nodes…





One more thing… write masks: $\langle \mathbf{m} \rangle$

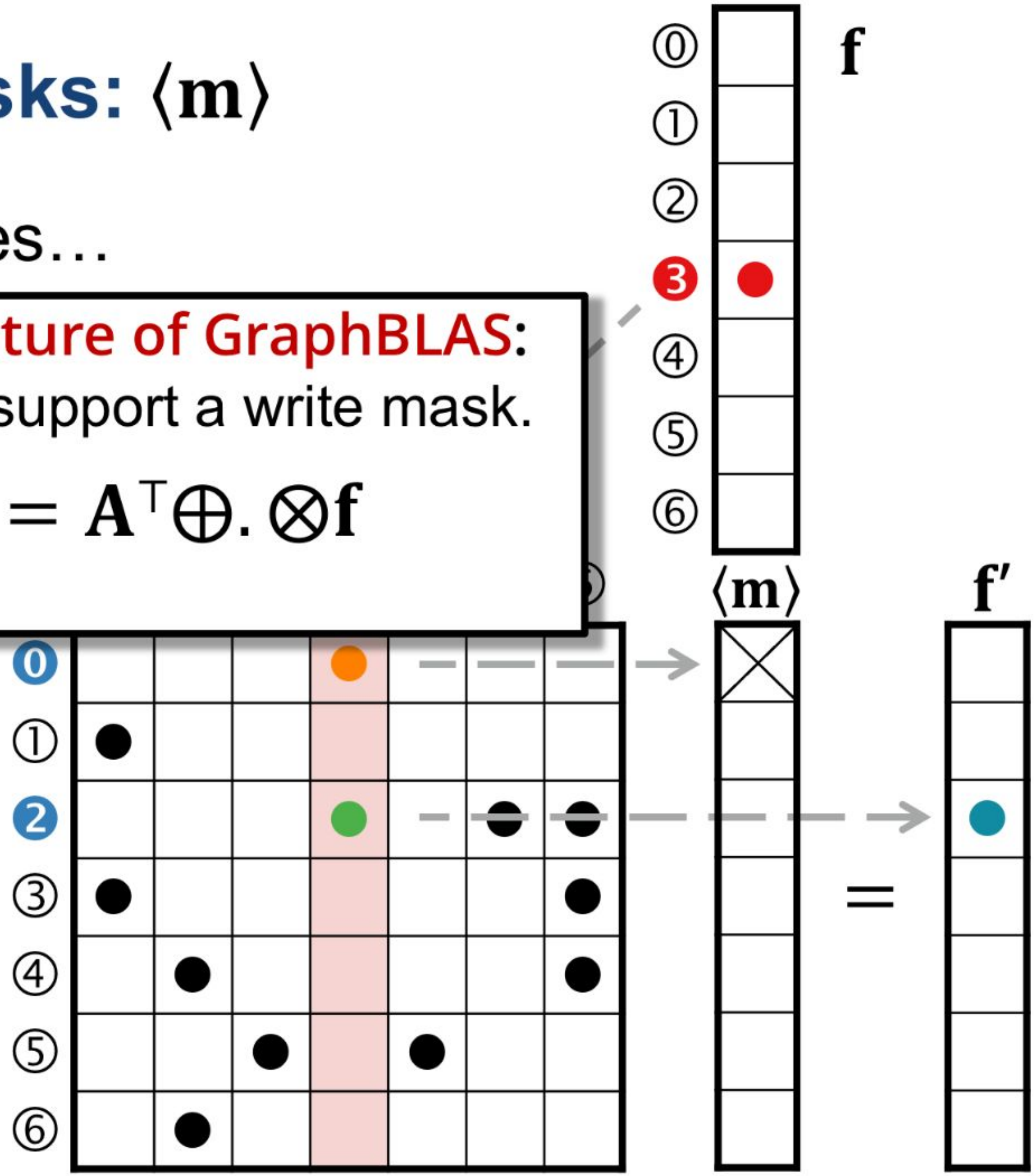
Often not interested in some nodes…



ANOTHER feature of GraphBLAS:

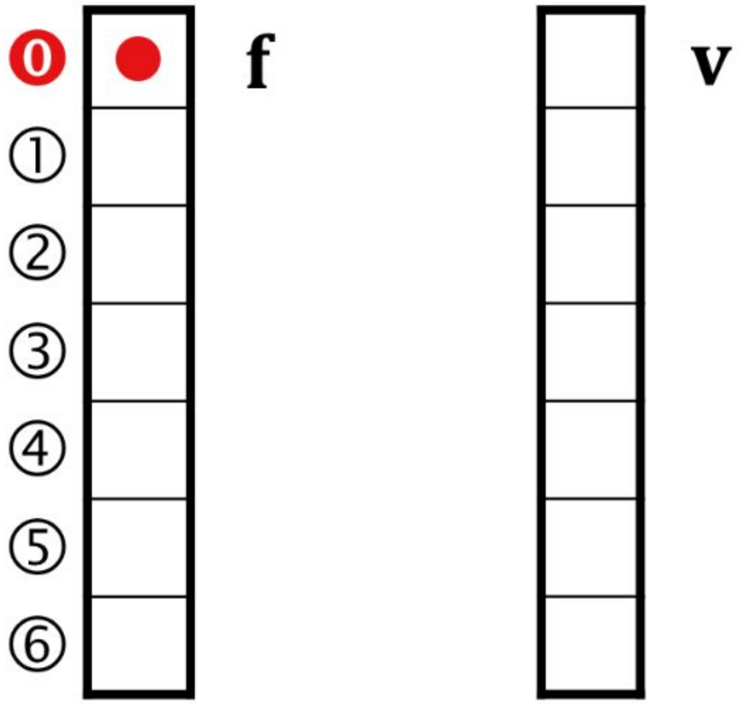
All operations support a write mask.

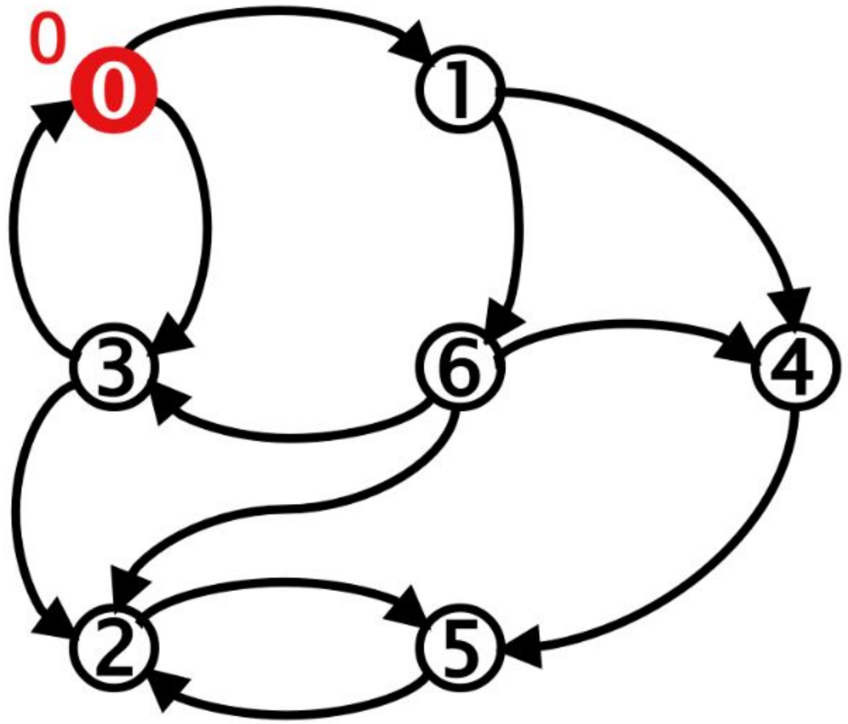
$$\mathbf{f}'\langle \mathbf{m} \rangle = \mathbf{A}^{\top} \oplus . \otimes \mathbf{f}$$



Example: Breadth-First Search (levels)

$\mathbf{f}(src) = \bullet$

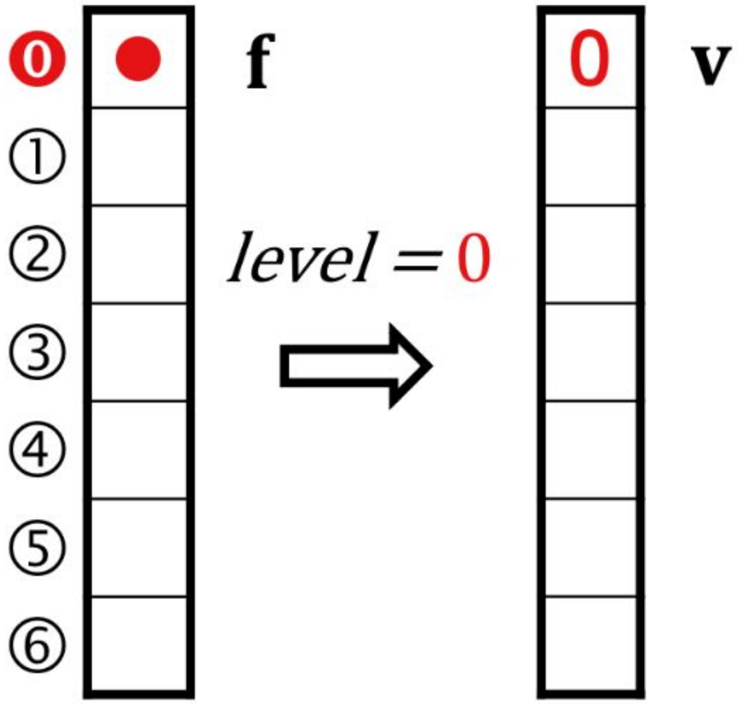


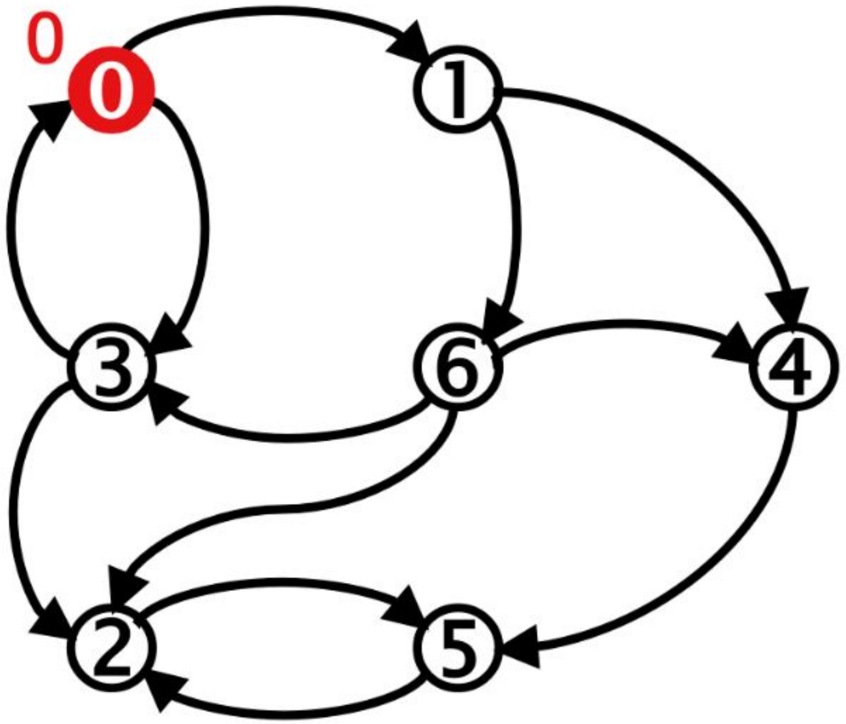


$\mathbf{A}^{\mathsf{T}}$	0	1	2	3	4	5	6
0				●			
1	●						
2				●		●	●
3	●						●
4		●					●
5			●		●		
6		●					

Example: Breadth-First Search (levels)

$$level = 0$$
$$\mathbf{v} \mathrel{+}= level * \mathbf{f}$$



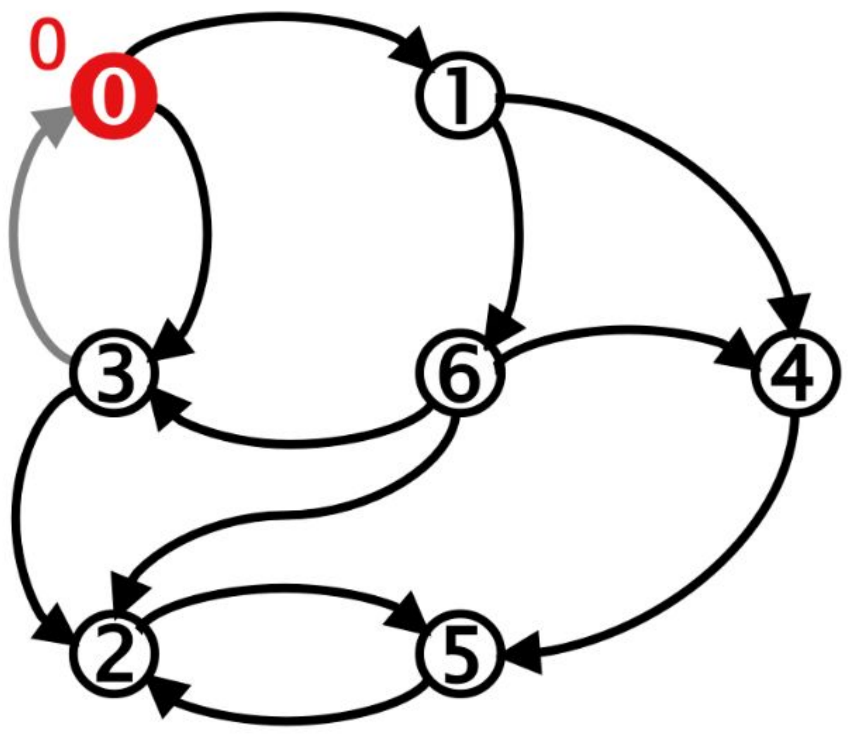


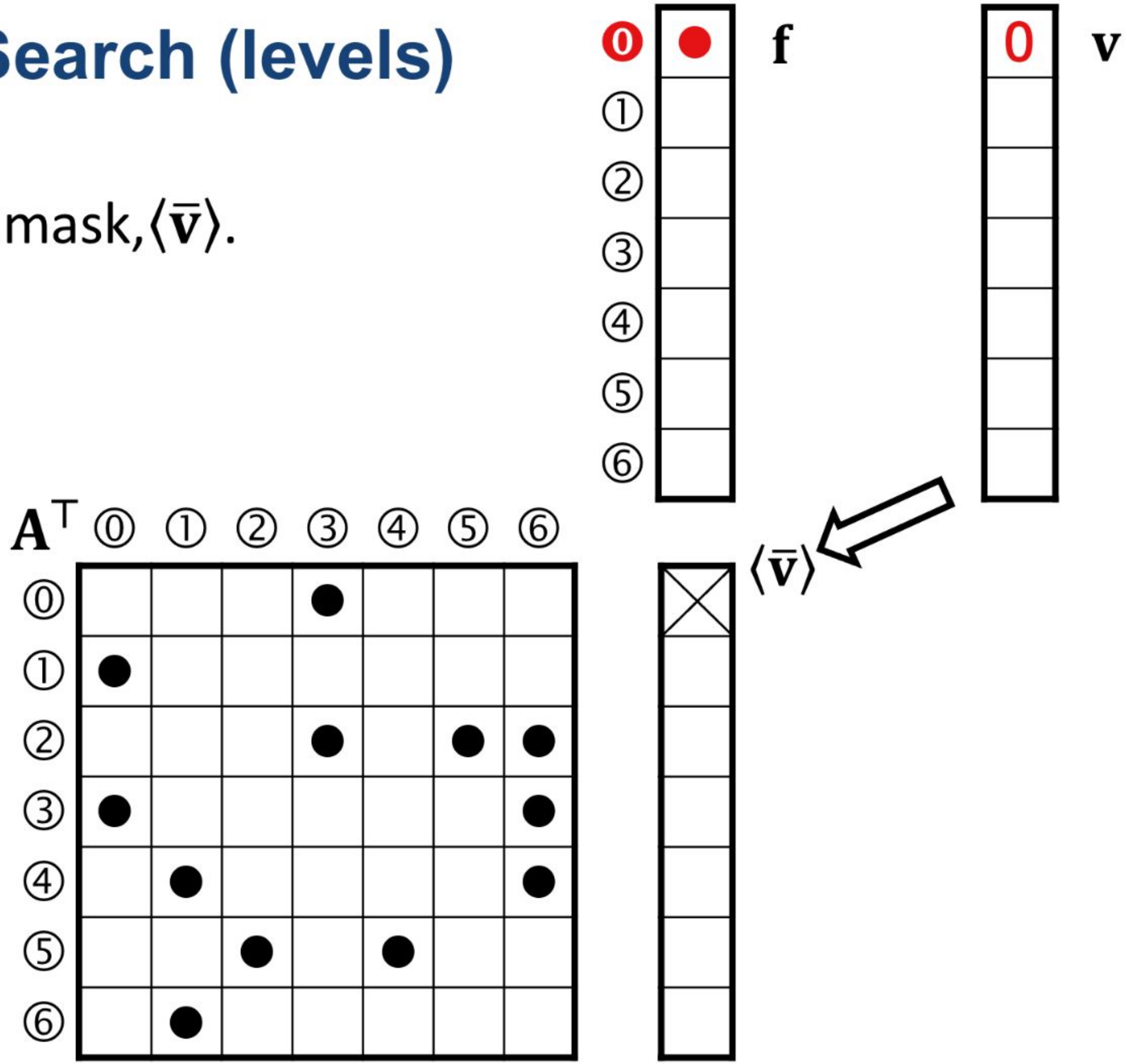
A^T	0	1	2	3	4	5	6
0				●			
1	●						
2				●		●	●
3	●						●
4		●					●
5			●		●		
6		●					

Example: Breadth-First Search (levels)

$level = 0$

$\mathbf{v} \mathrel{+}= level \ast \mathbf{f}$ // Use $\mathbf{v}$ as a mask, $\langle \bar{\mathbf{v}} \rangle$.



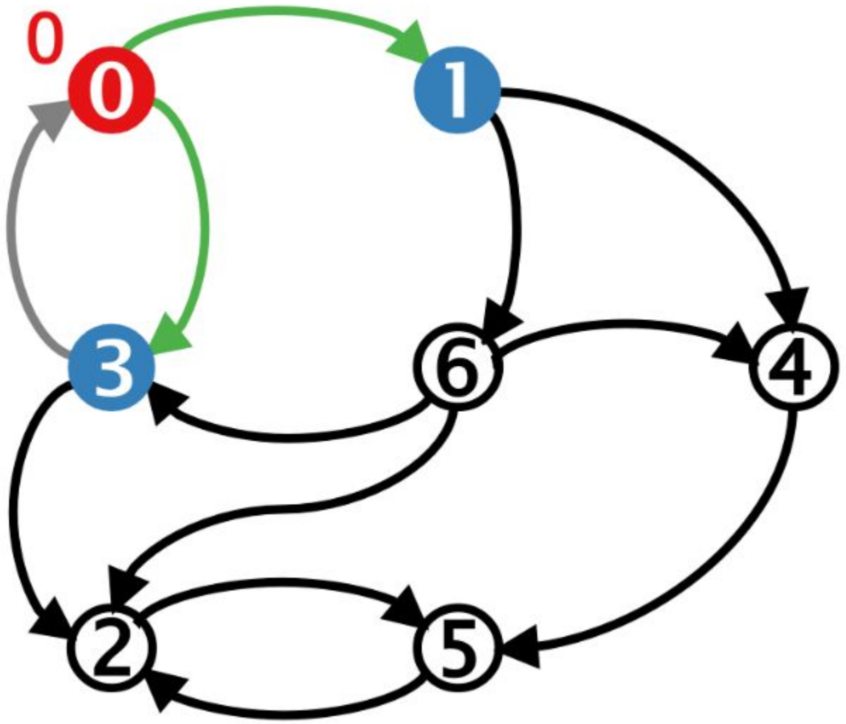


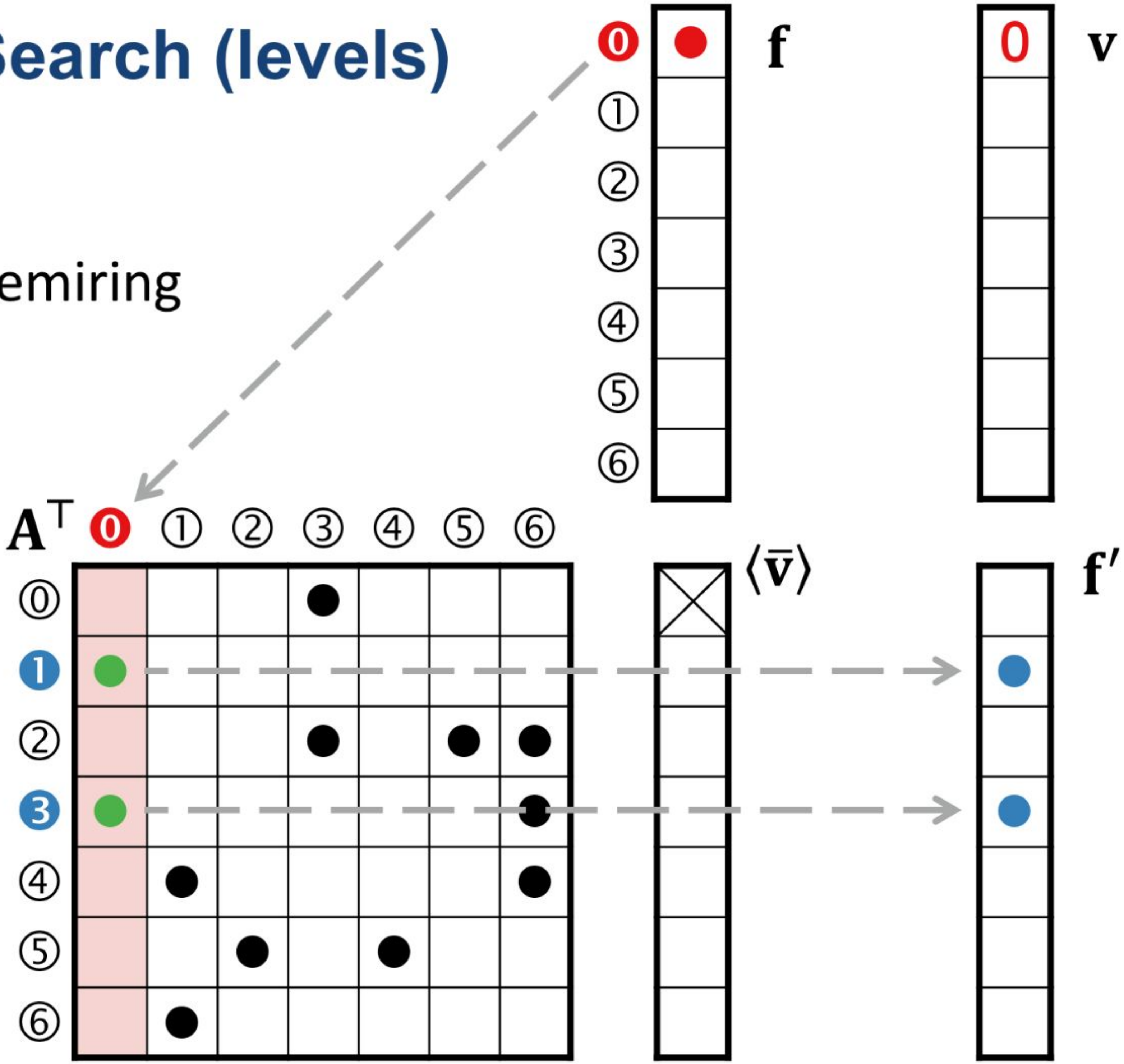
Example: Breadth-First Search (levels)

$$level = 0$$

$$\mathbf{v} \mathrel{+}= level * \mathbf{f}$$

$$\mathbf{f}'\langle\bar{\mathbf{v}}\rangle = \mathbf{A}^{\top} \oplus . \otimes \mathbf{f} \quad \text{// Boolean semiring}$$





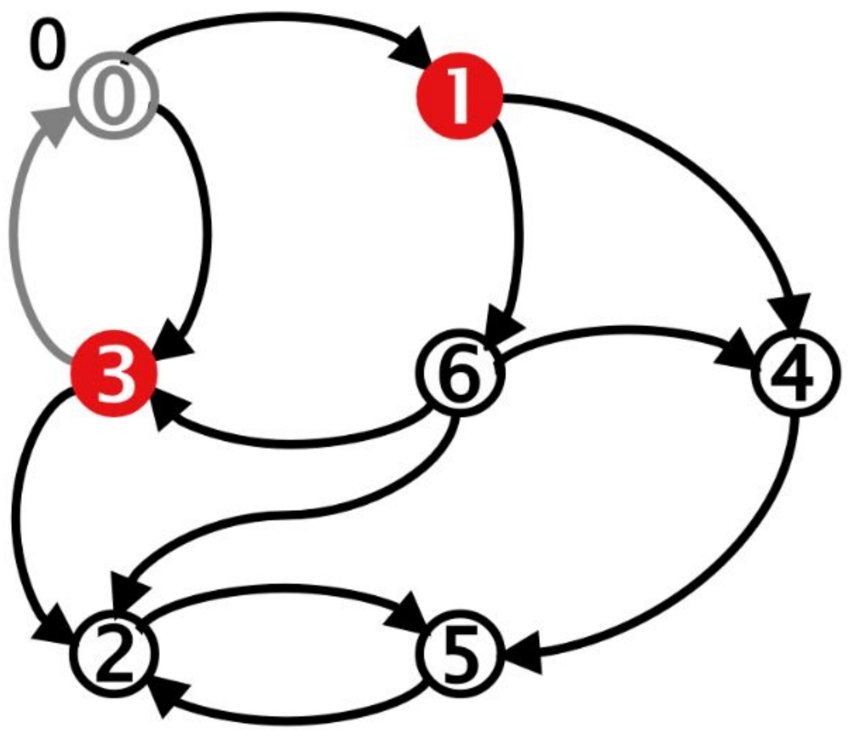
Example: Breadth-First Search (levels)

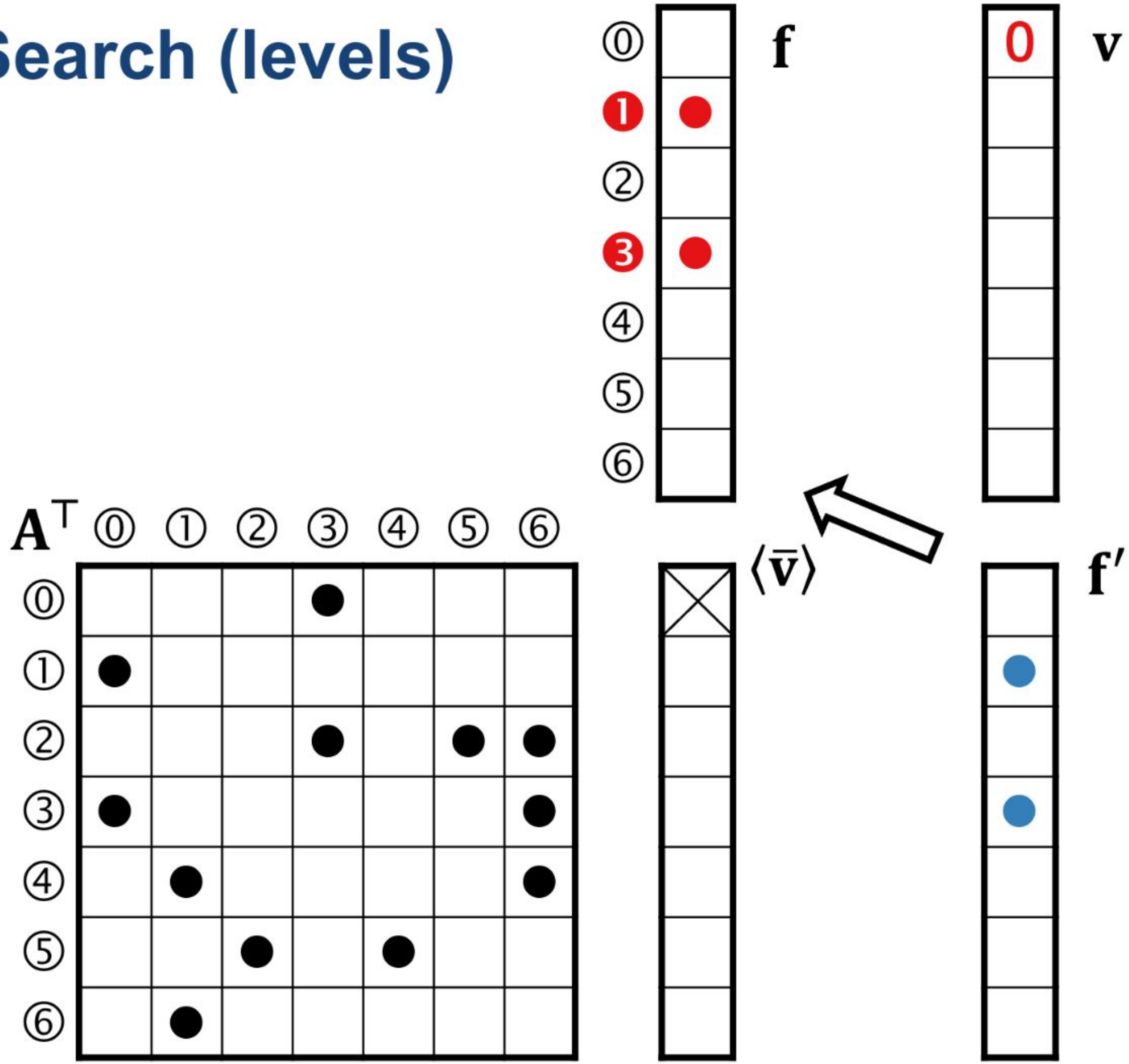
$$level = 0$$

$$\mathbf{v} \mathrel{+}= level * \mathbf{f}$$

$$\mathbf{f}'\langle \bar{\mathbf{v}} \rangle = \mathbf{A}^{\mathsf{T}} \oplus . \otimes \mathbf{f}$$

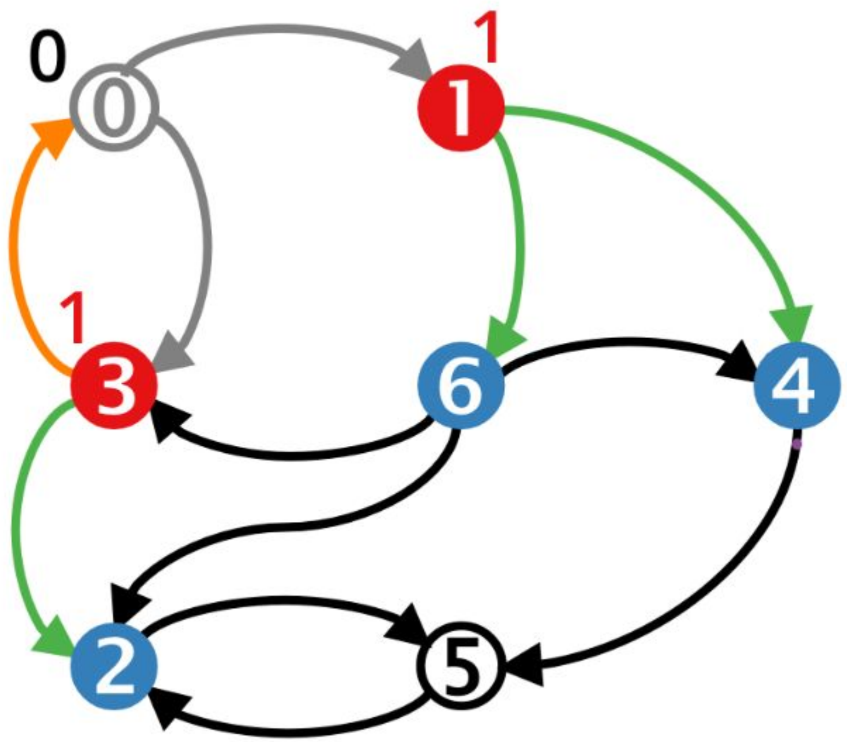
$$\mathbf{f} = \mathbf{f}'$$

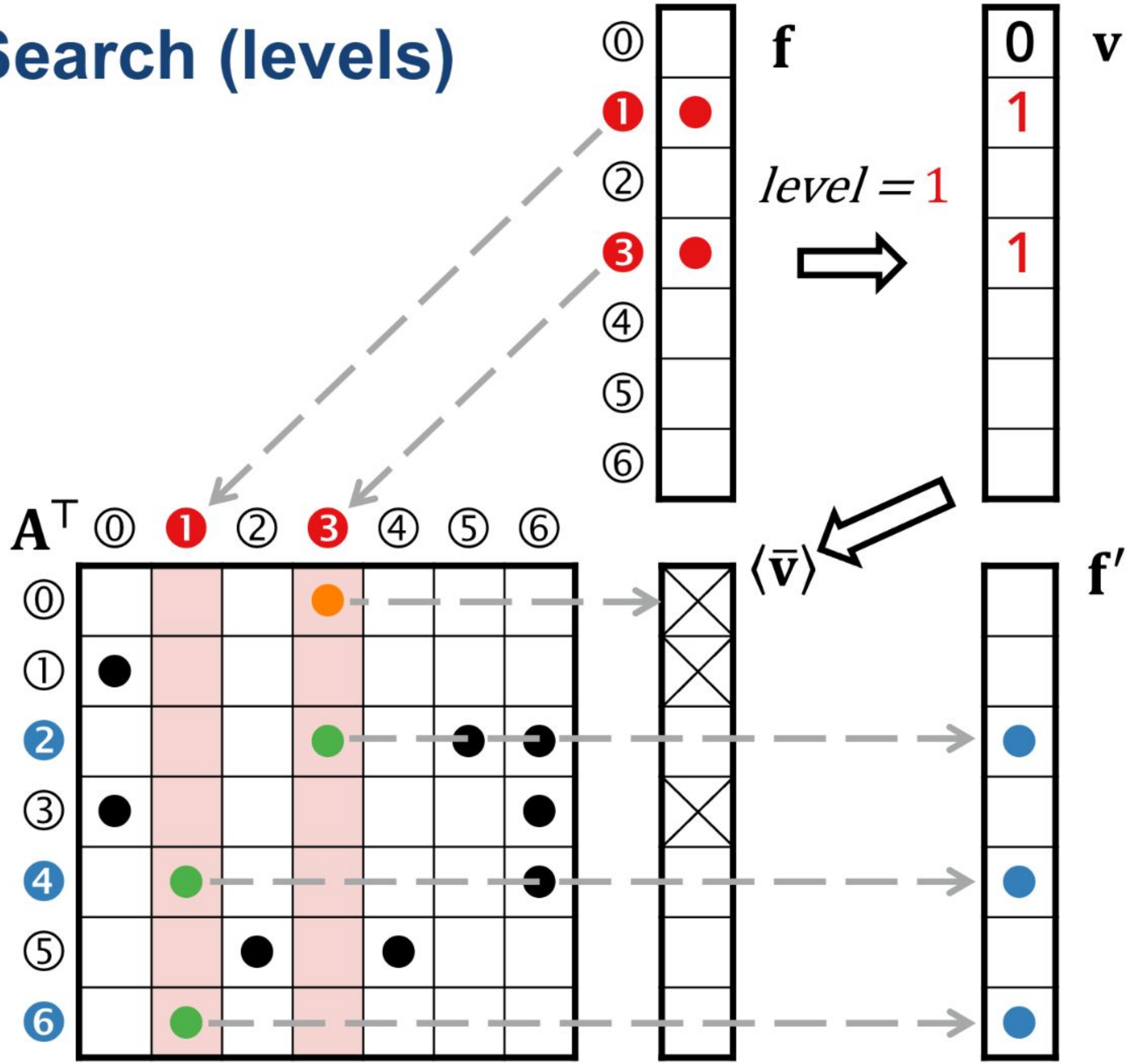




Example: Breadth-First Search (levels)

$$level = 1$$
$$\mathbf{v} \mathrel{+}= level * \mathbf{f}$$
$$\mathbf{f}'\langle\bar{\mathbf{v}}\rangle = \mathbf{A}^{\top} \oplus . \otimes \mathbf{f}$$
$$\mathbf{f} = \mathbf{f}'$$





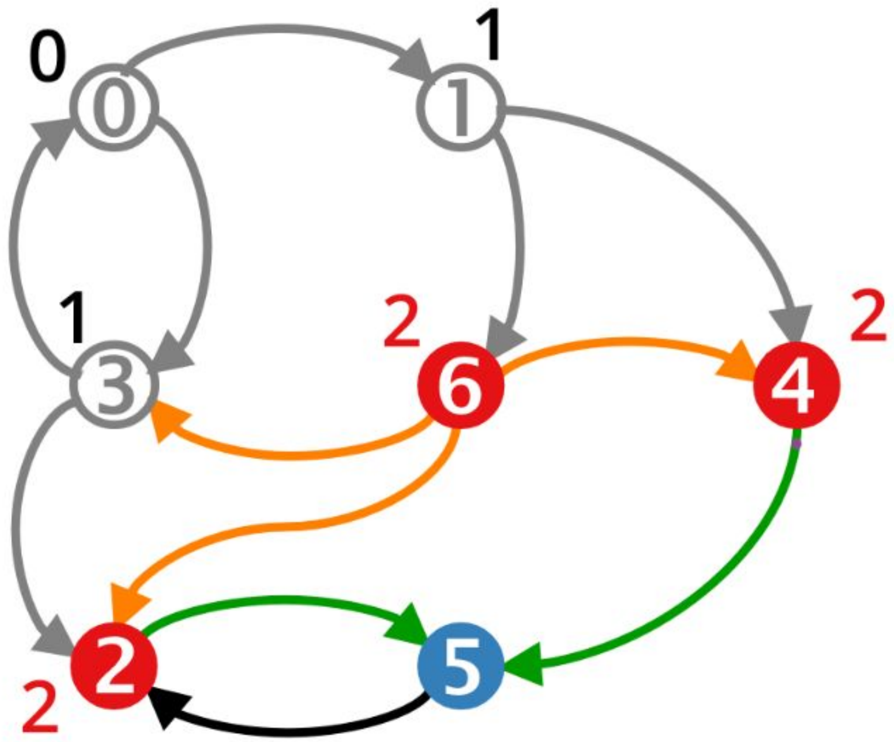
Example: Breadth-First Search (levels)

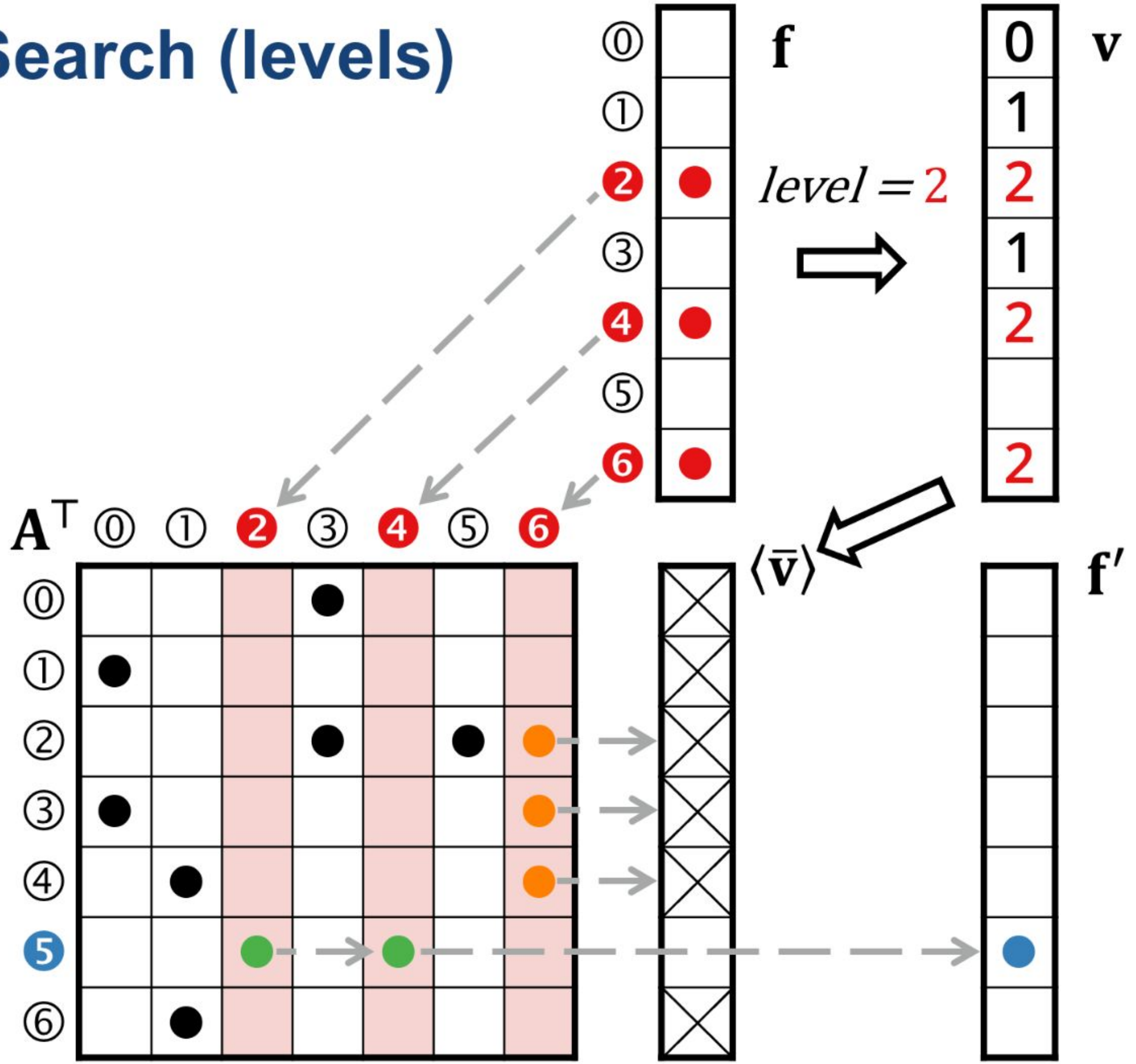
$$level = 2$$

$$\mathbf{v} \mathrel{+}= level * \mathbf{f}$$

$$\mathbf{f}'\langle\bar{\mathbf{v}}\rangle = \mathbf{A}^{\top} \oplus.\otimes \mathbf{f}$$

$$\mathbf{f} = \mathbf{f}'$$





Example: Breadth-First Search (levels)

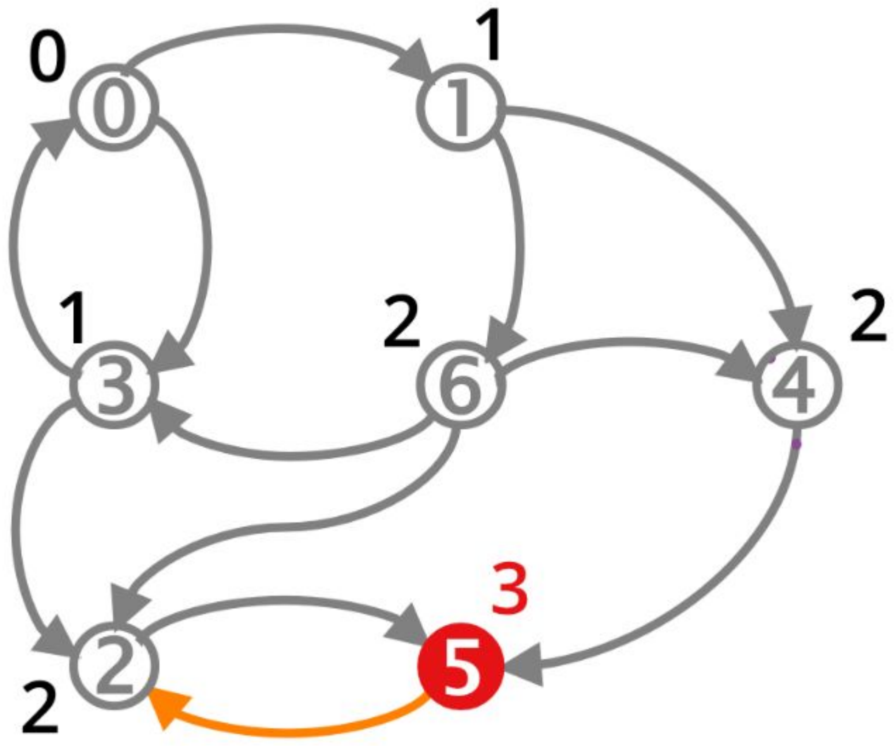
$level = 3$

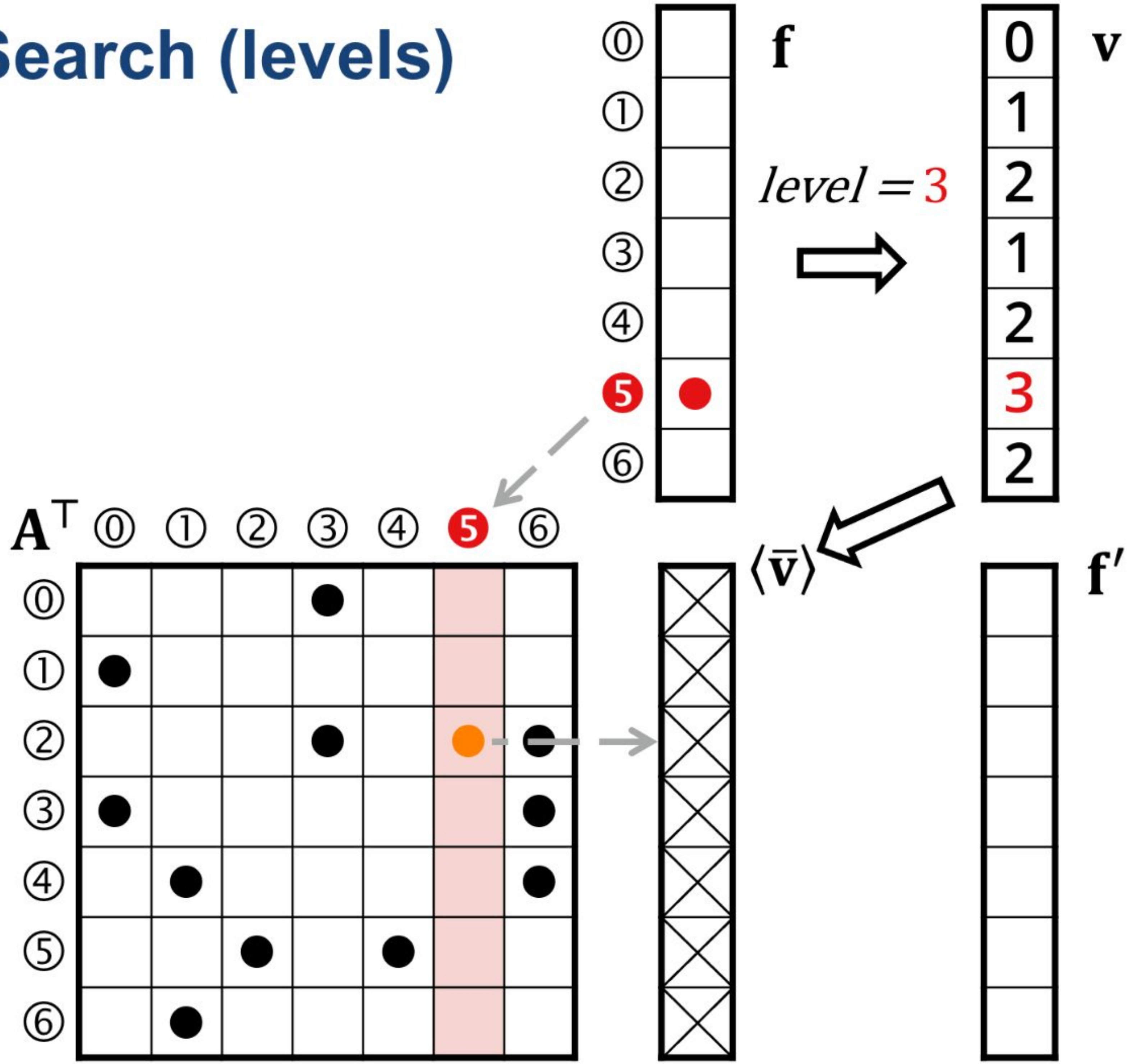
$\mathbf{v} \mathrel{+}= level * \mathbf{f}$

$\mathbf{f}'\langle \bar{\mathbf{v}} \rangle = \mathbf{A}^{\mathsf{T}} \oplus.\otimes \mathbf{f}$

$\mathbf{f} = \mathbf{f}'$

if $\mathbf{f}$.empty() return v





Example: Breadth-First Search (levels)

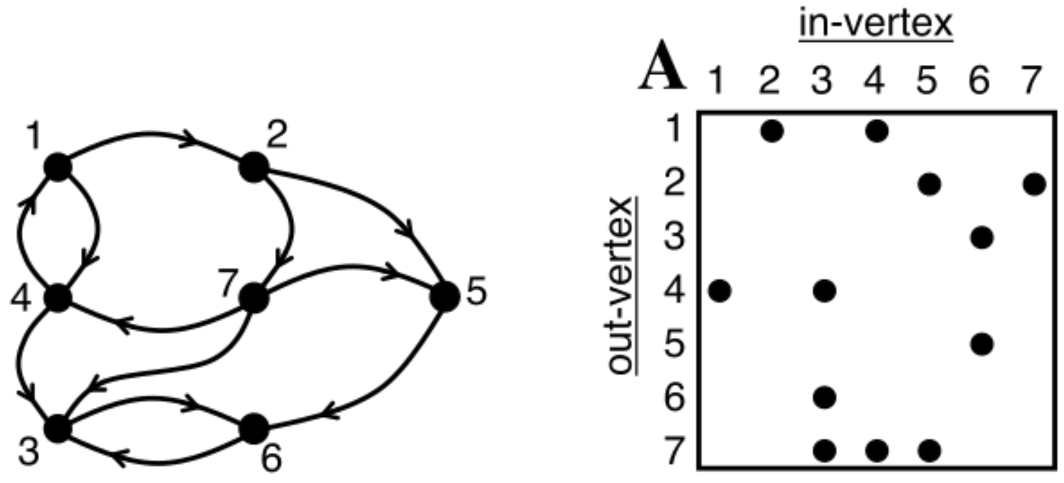
- **Input:** adjacency matrix $\mathbf{A}$ (Boolean), source vertex src (integer)
- **Output:** visited vertices vector, $\mathbf{v}$ (integer)
- **Workspace:** frontier vector $\mathbf{f}$ (Boolean)

1. $\mathbf{f}(src) = \text{true}$
2. $level = 0$
3. while $!\,\mathbf{f}.\text{empty}()$
4. $\quad \mathbf{v} \mathrel{+}= level * \mathbf{f}$
5. $\quad \mathbf{f}\langle \bar{\mathbf{v}} \rangle = \mathbf{A}^{\top} \oplus . \otimes \mathbf{f}$ `// using the Boolean semiring (OR.AND)`
6. $\quad ++level$

Prior work: GraphBLAS C API and Onwards

GraphBLAS C API

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

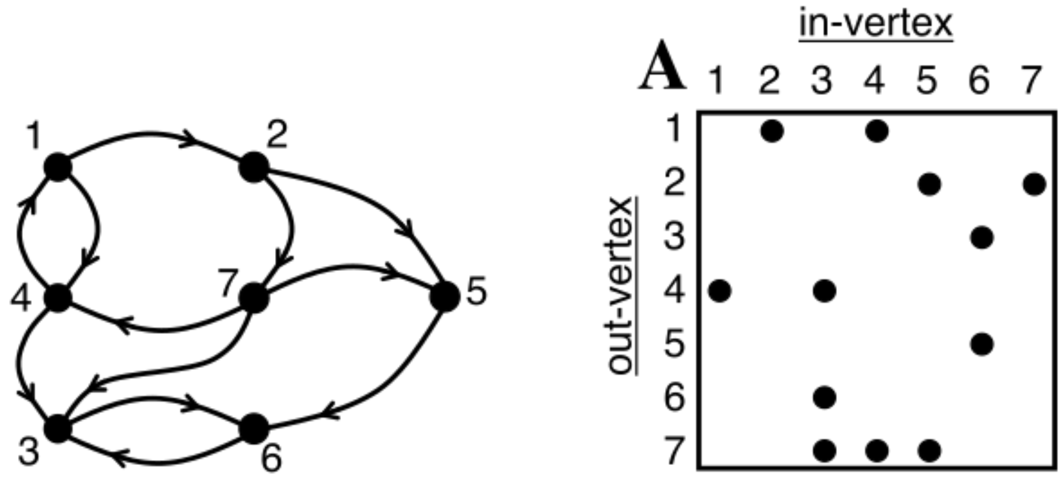


- Revolve around sparse matrix and vector operations which can use **arbitrary semirings** instead of classical (+, *)

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

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

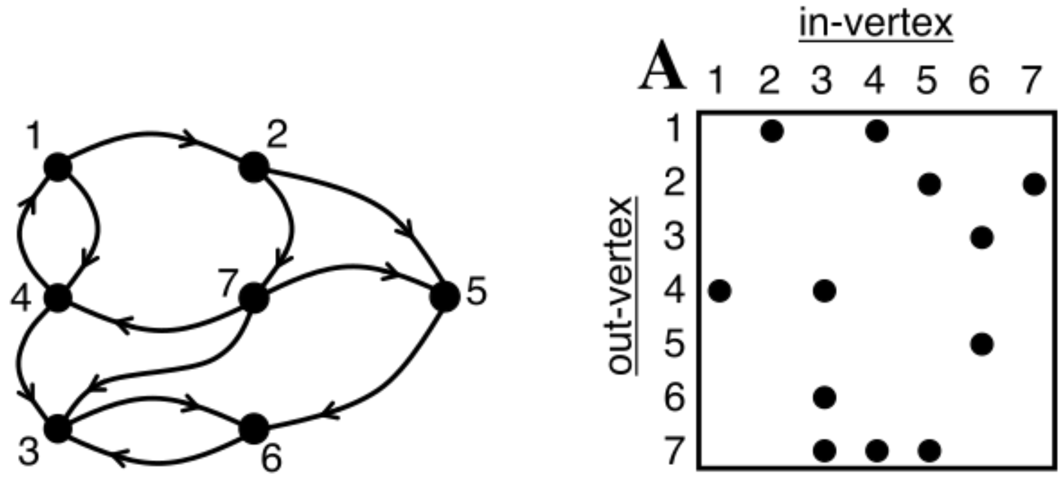
GraphBLAS C API



- Provides **uniform API** for **graph algorithms** in the **language of linear algebra**
- Revolve around sparse matrix and vector operations which can use **arbitrary semirings** instead of classical (+, *)
- Current version of C API spec. is 1.3 (**2.0 arriving imminently!**)
- C offers great **portability** (Python, bindings, etc.), but has some **disadvantages**...

GraphBLAS C API

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



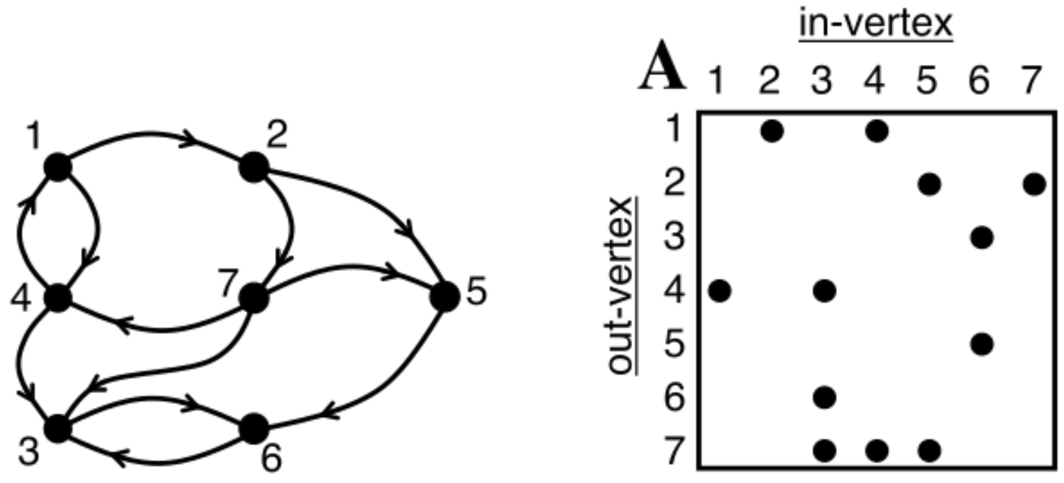
- Revolve around sparse matrix and vector operations which can use **arbitrary semirings** instead of classical (+, *)

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

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

GraphBLAS C API

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



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

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

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

The Problem with Types...

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

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

- **Combinatorial explosion**

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

...

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

The Problem with Types...

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

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

- **Combinatorial explosion**

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

...

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

The Problem with Types...

- If you're familiar with the (C)**BLAS**, there is a function for each **scalar type**
- GraphBLAS supports a wide variety of **scalar types** and **binary operators**
- **Combinatorial explosion**

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

...

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

C API: Quality of Life Issues

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

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

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

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

C API: Quality of Life Issues

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

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

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

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

C API: Quality of Life Issues

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

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

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

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

C API: Quality of Life Issues

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

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

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

Descriptors

Flags for transformations

Transpose, complement, structure-only, etc.

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

C API: Quality of Life Issues

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

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

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

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

- Users have already run into cases where they wish to use **more complex types**.

C API: Quality of Life Issues

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

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

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

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

- Users have already run into cases where they wish to use **more complex types**.

C API: Quality of Life Issues

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

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

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

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

- Users have already run into cases where they wish to use **more complex types**.

C API: Issues with Types

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

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

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

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

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

```
GrB_apply(C, ..., my_scale_2, A, desc);
```

C API: Issues with Types

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

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

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

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

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

```
GrB_apply(C, ..., my_scale_2, A, desc);
```

Drafting a GraphBLAS C++ API

C++ Has a Rich Type System

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

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

- Things like **views** can simplify APIs

C++ Has a Rich Type System

- User-defined types are **first-class types**
- They simply need be **copy constructible, etc.**
- Things like **views** can simplify APIs

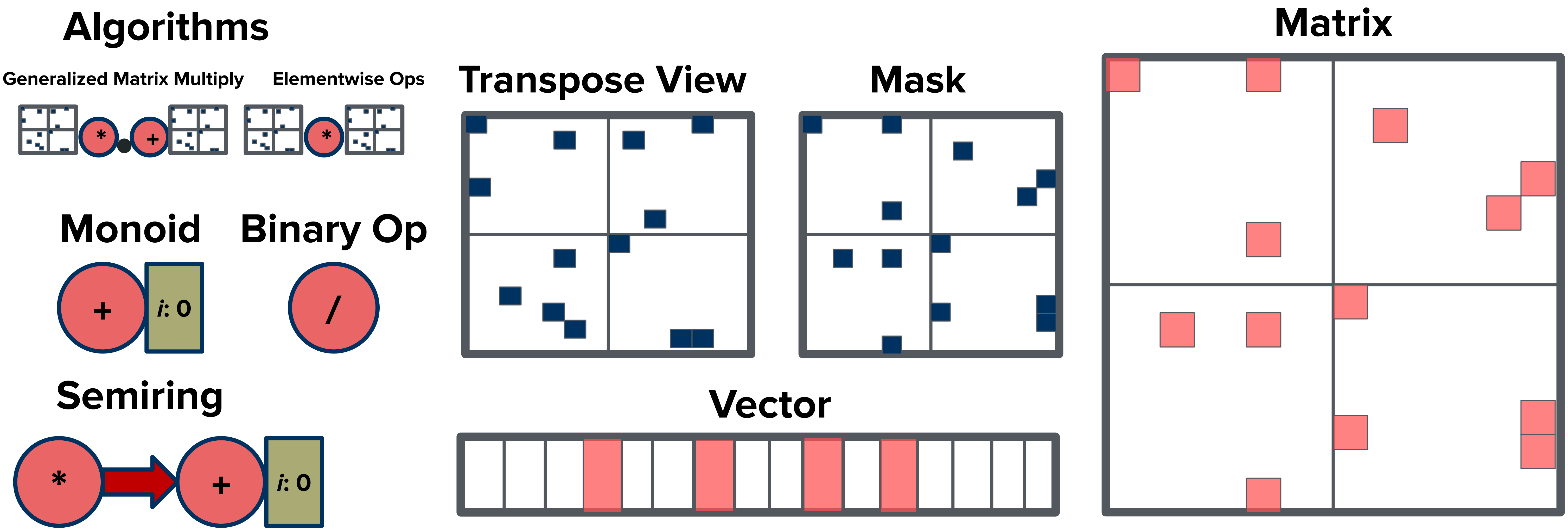
C++ Has a Rich Type System

- User-defined types are **first-class types**
- They simply need be **copy constructible, etc.**
- Things like **views** can simplify APIs

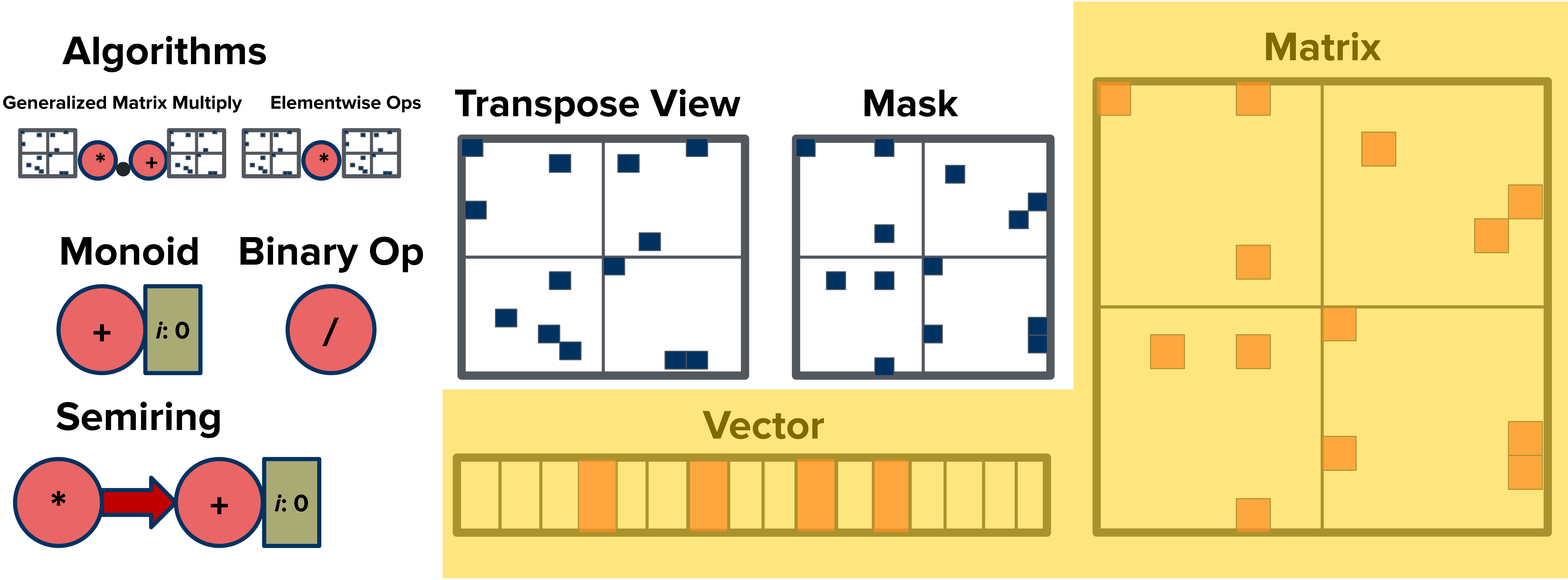
Disclaimer: API in Progress

- The GraphBLAS C++ API is still in **draft** process
- Specific **names** and **APIs** may change
- There are currently two **draft implementations**, GBTL and RGRI
- Some slide contents **may be in RGRI**, but not necessarily in C++ spec (yet)

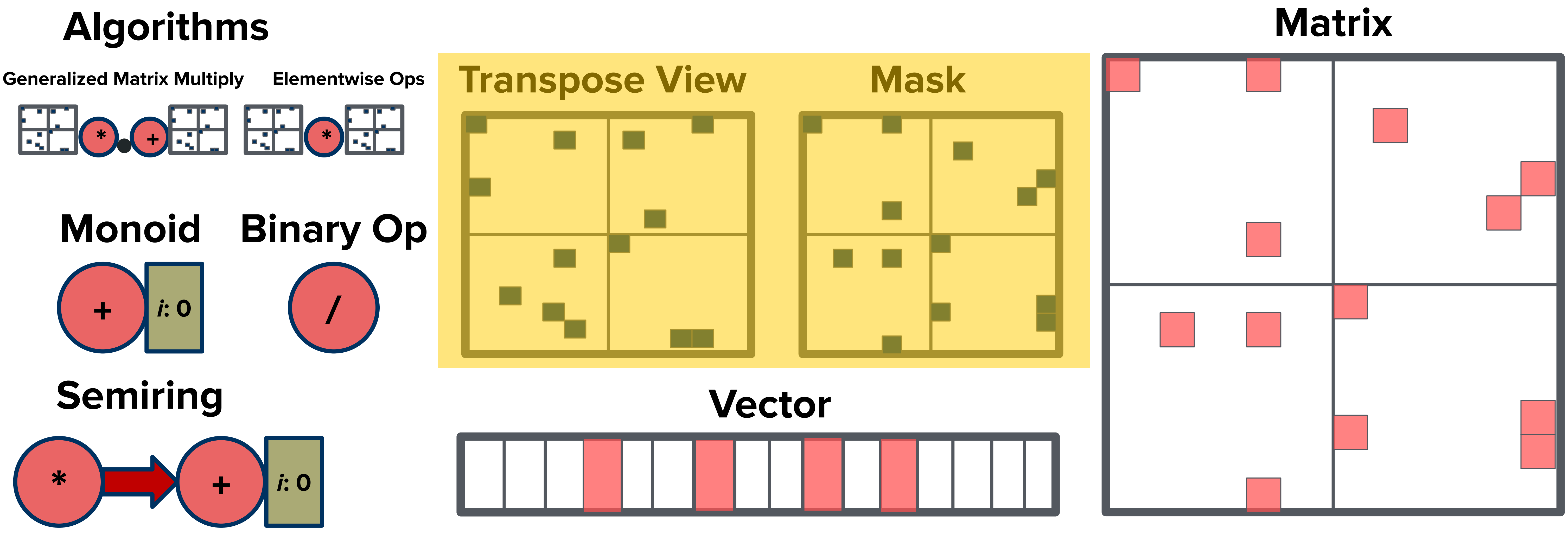
GraphBLAS Concepts



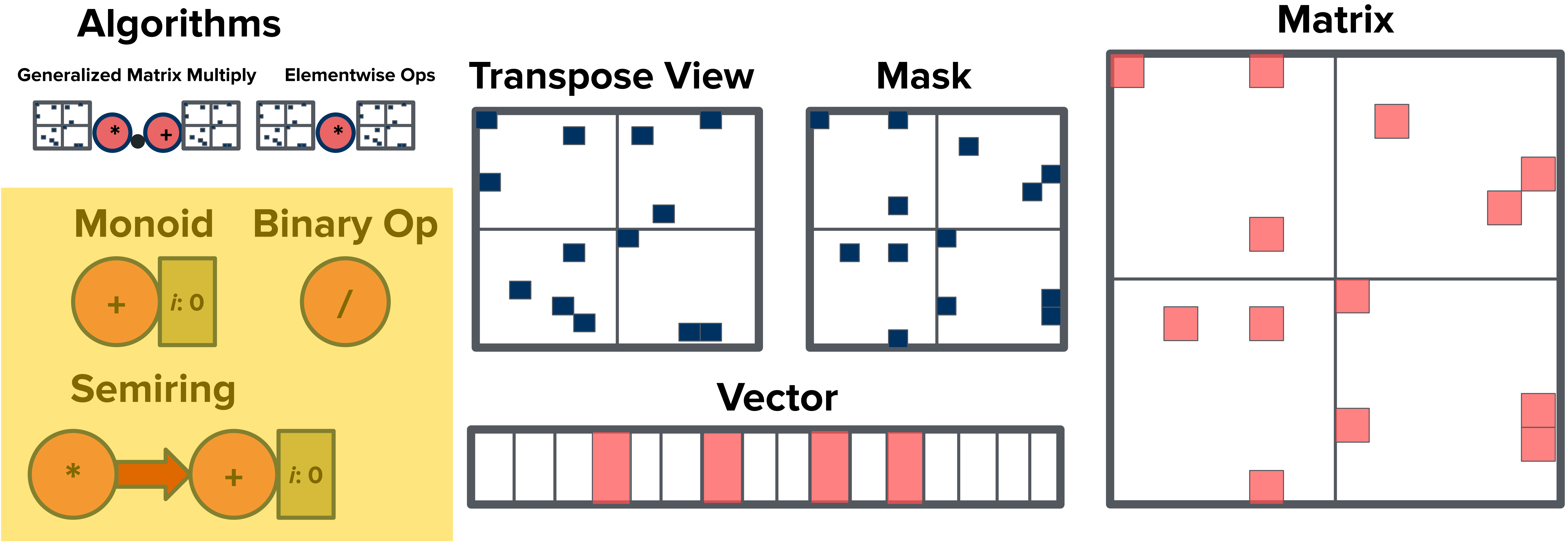
GraphBLAS Concepts



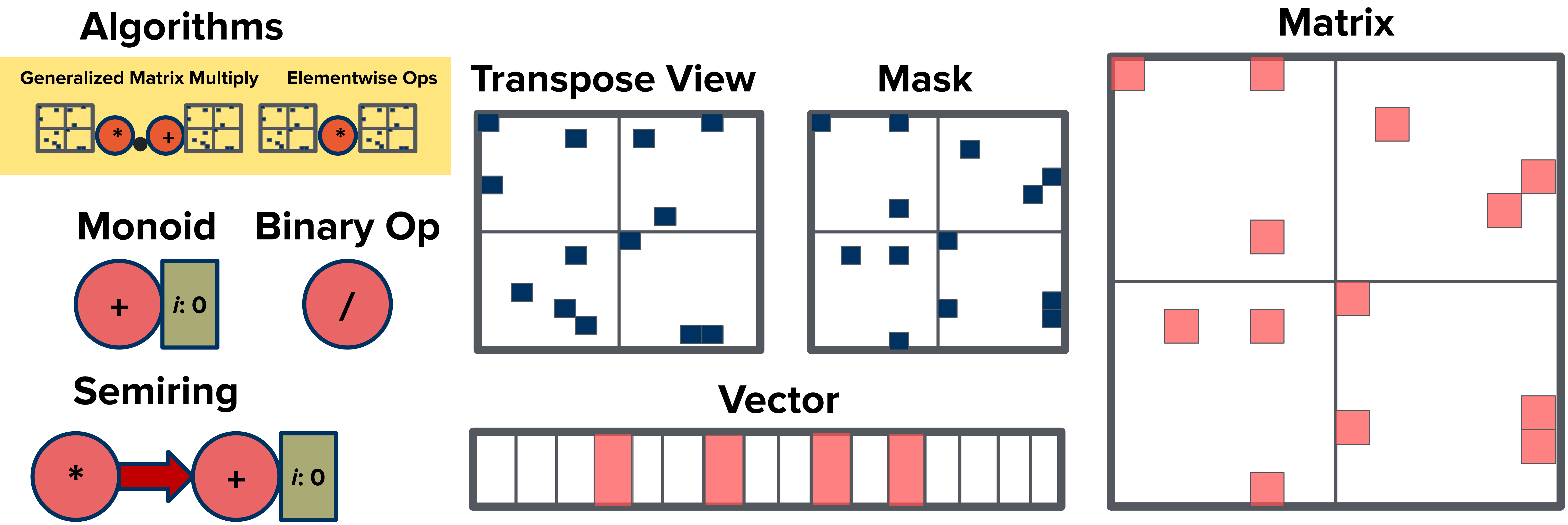
GraphBLAS Concepts



GraphBLAS Concepts



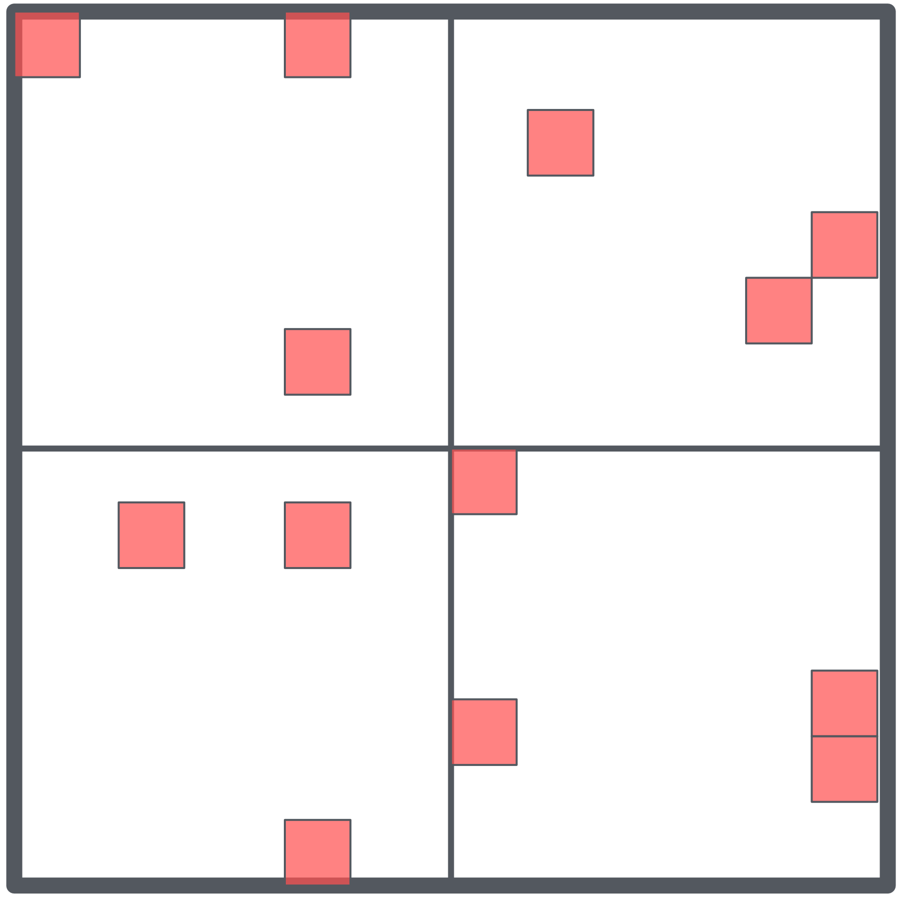
GraphBLAS Concepts



GraphBLAS Matrix

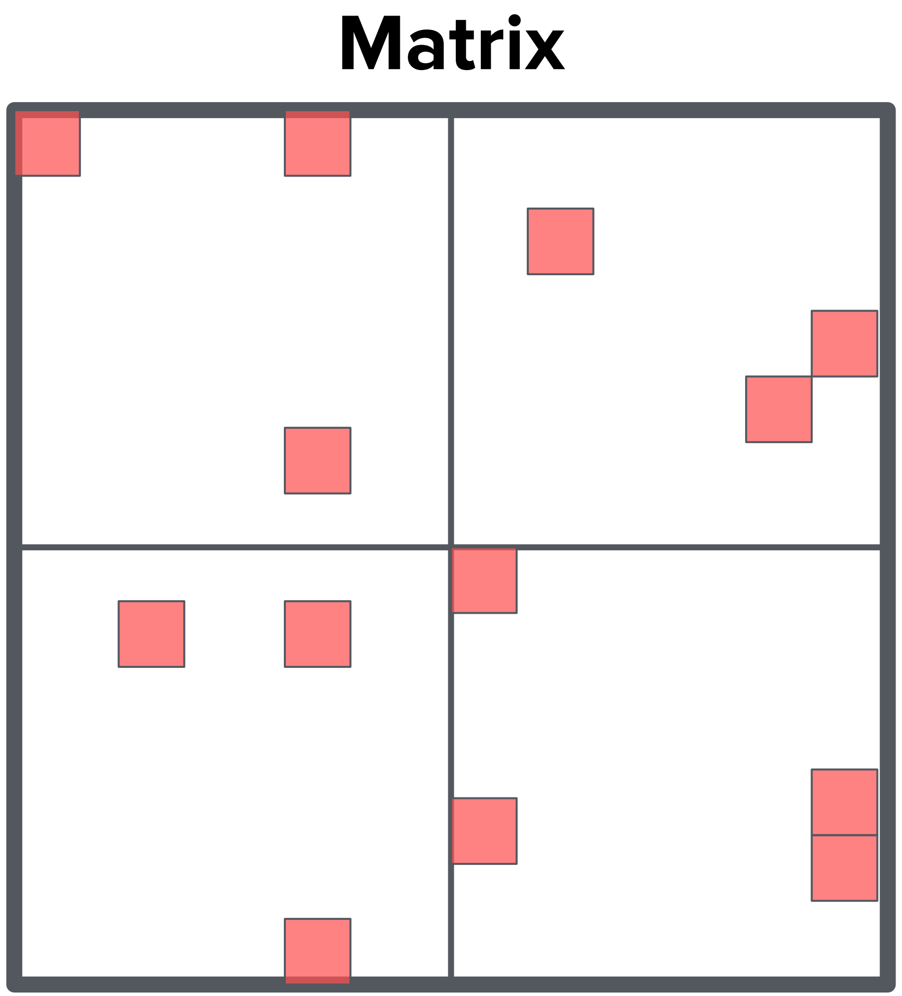
- A matrix is a collection of **stored values**
- It has a **shape** (number of rows, cols)
- It has a **size** (number of stored values)
- Can **access individual locations**
- Can **iterate** over values

Matrix



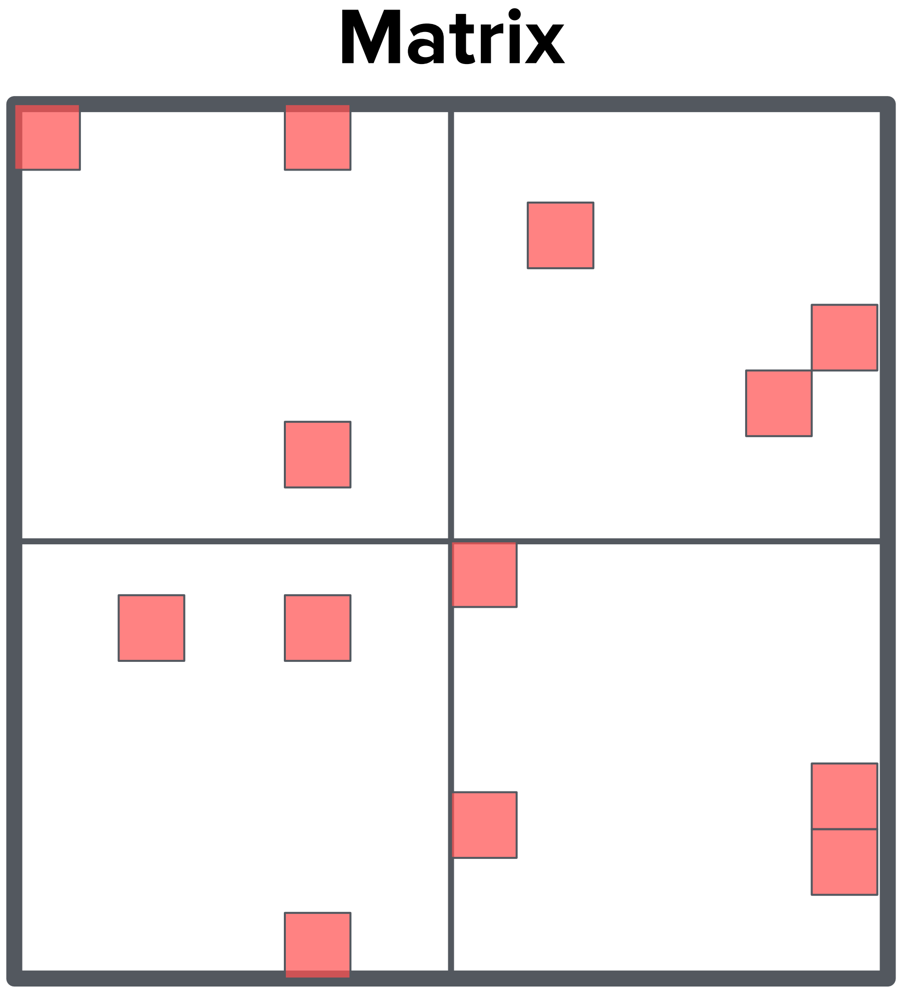
GraphBLAS Matrix

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



GraphBLAS Matrix

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

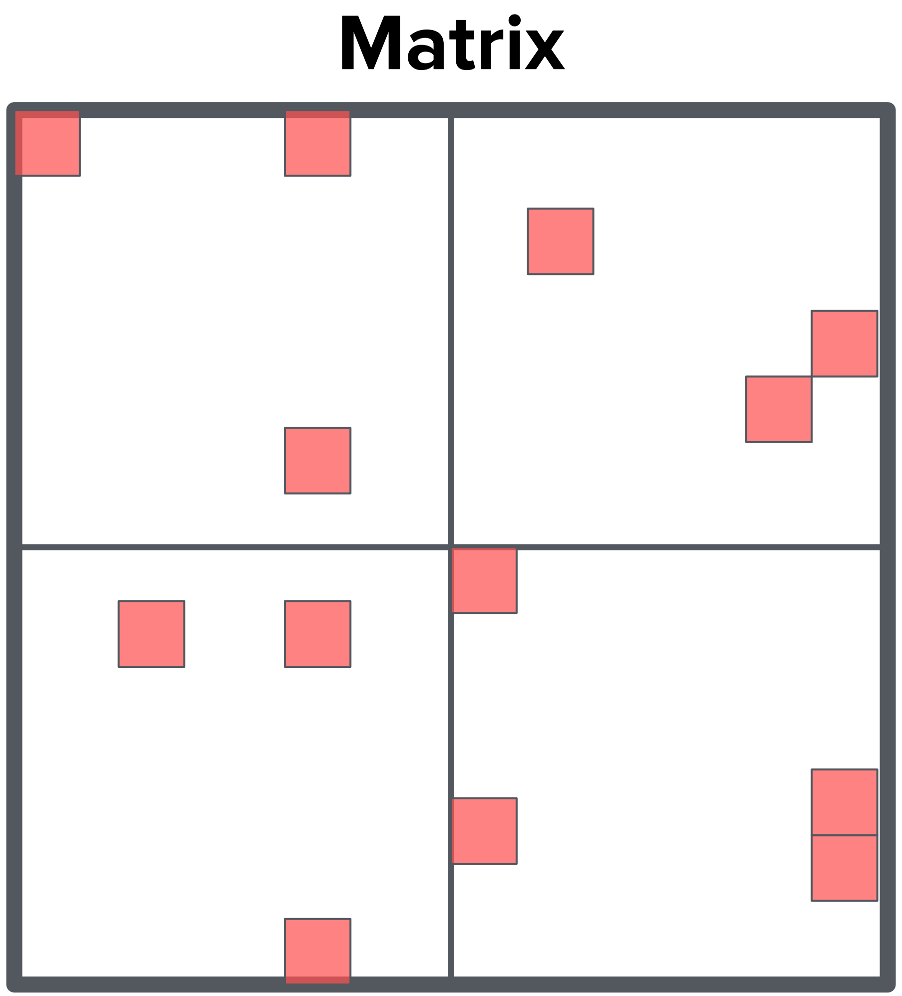


Carnegie Mellon University
Software Engineering Institute

Berkeley
UNIVERSITY OF CALIFORNIA

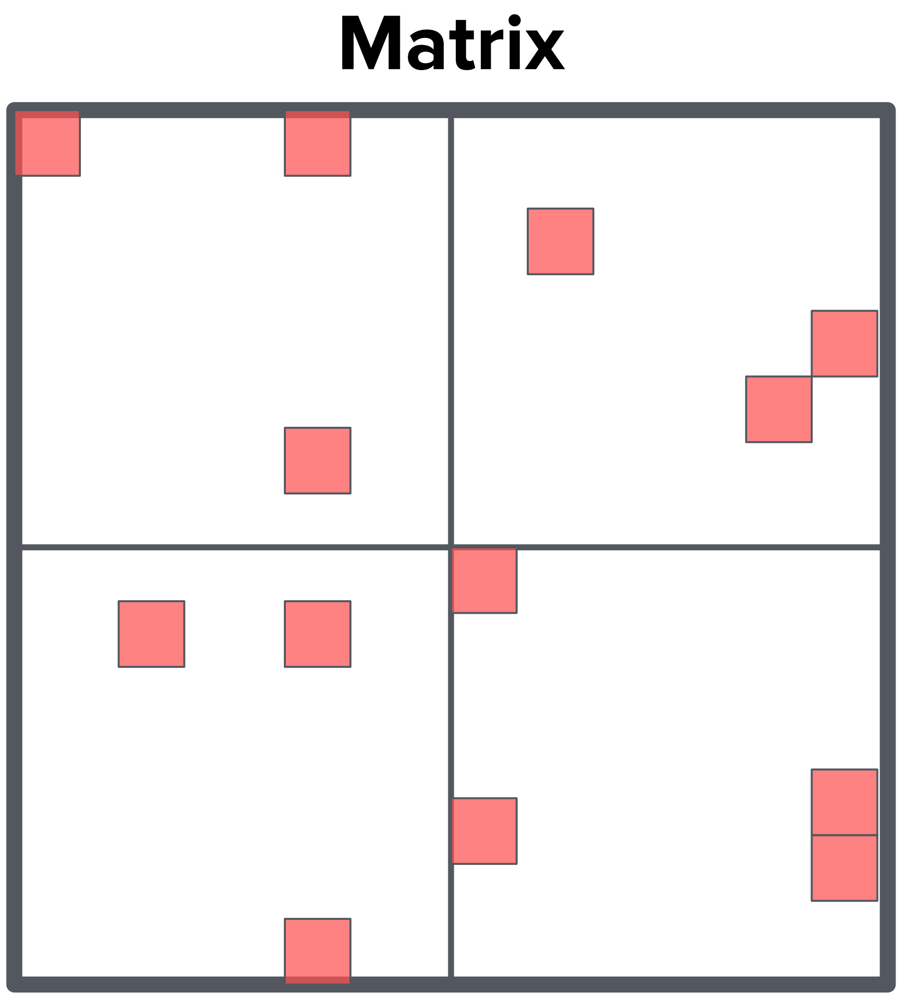
GraphBLAS Matrix

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



GraphBLAS Matrix

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

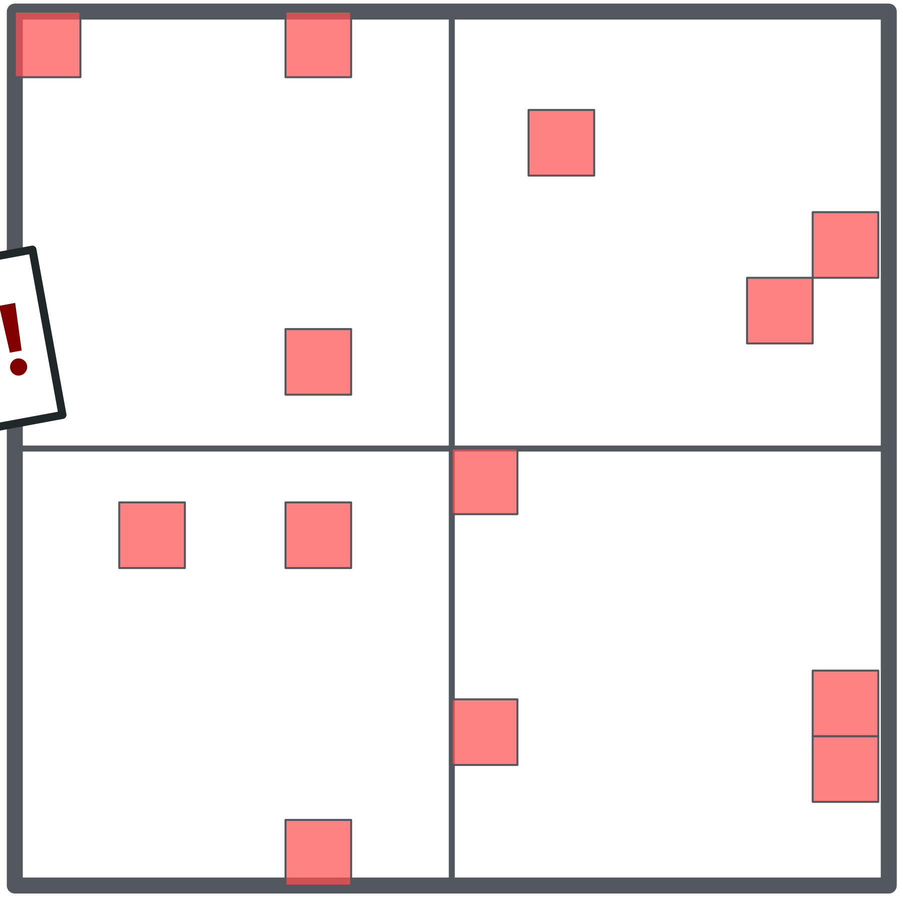


GraphBLAS Matrix

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

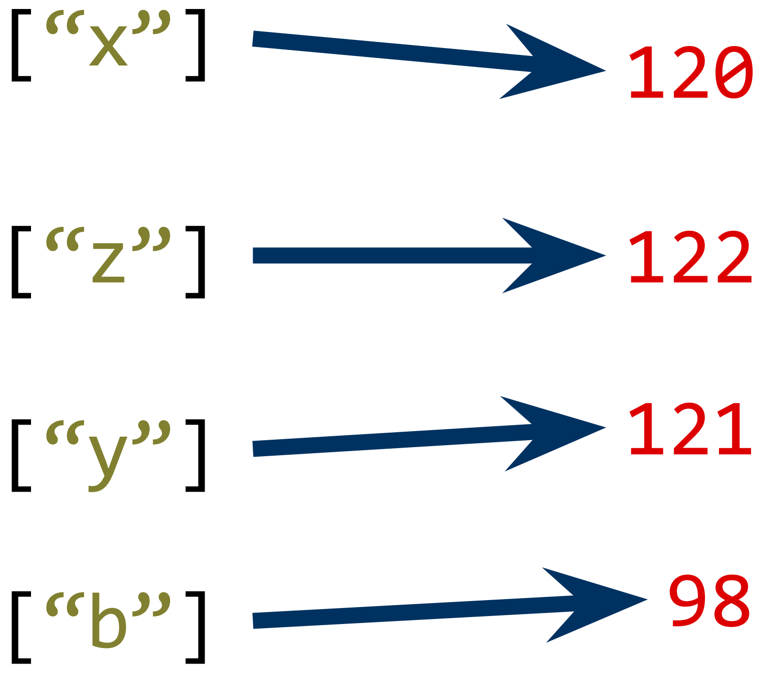
Not included: implicit zero value!

Matrix



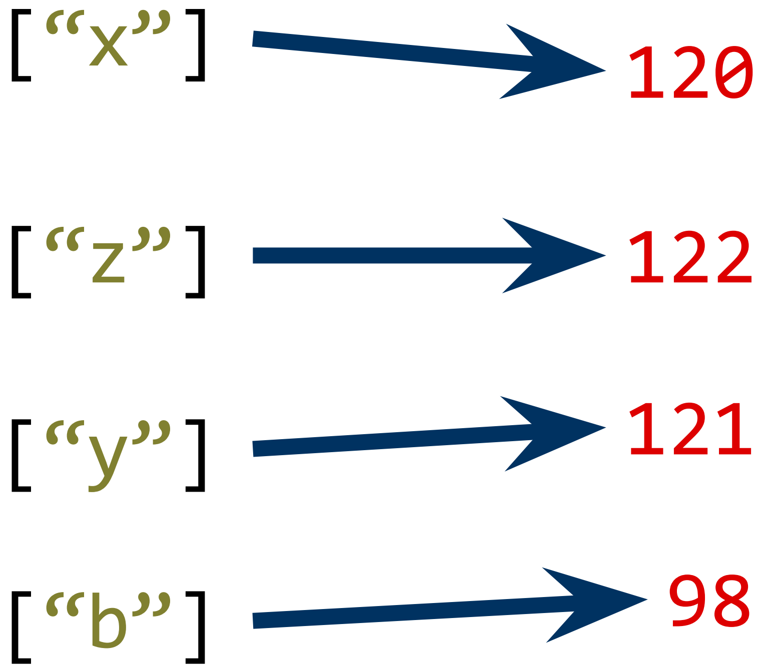
Sparse Matrix - Similarities to `std::unordered_map`

- Distinct **set of keys**
- Each key **associated** with **a value**
- Individual **lookup/insertion** by key
- **Iteration** over **unordered range of values**



Sparse Matrix - Similarities to `std::unordered_map`

- Distinct **set of keys**
- Each key **associated** with **a value**
- Individual **lookup/insertion** by key
- **Iteration** over **unordered range of values**



Sparse Matrix - Similarities to `std::unordered_map`

- Distinct **set of keys**
- Each key **associated** with **a value**
- Individual **lookup/insertion** by key
- **Iteration** over **unordered range of values**

["x"] → 120

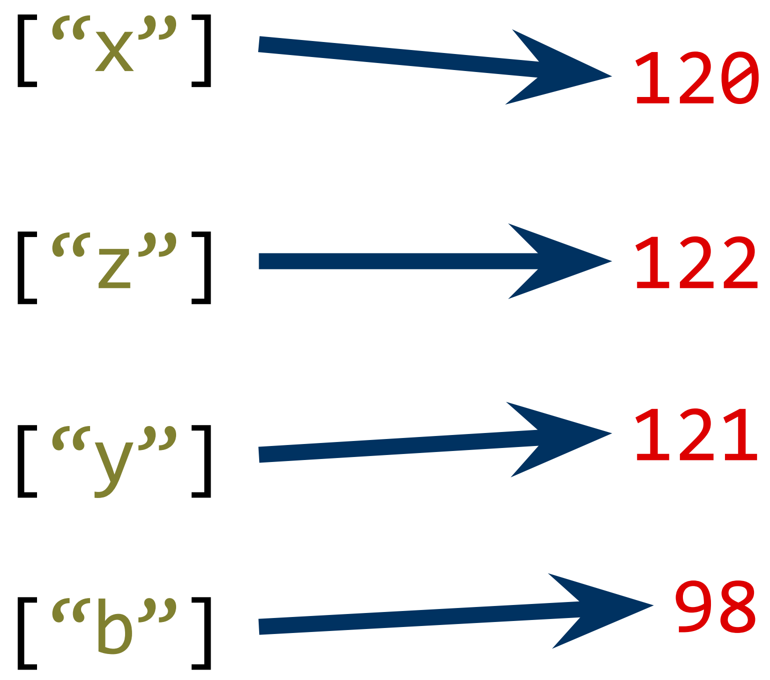
["z"] → 122

["y"] → 121

["b"] → 98

Sparse Matrix - Similarities to `std::unordered_map`

- Distinct **set of keys**
- Each key **associated** with **a value**
- Individual **lookup/insertion** by key
- **Iteration** over **unordered range of values**



Sparse Matrix - *Differences* from `std::unordered_map`

- **`key_type`** is **pair-like type** filled with **integral values**
- **Matrix shape** restricts **valid key values**
- **Implementation** will use **highly specialized sparse matrix formats**
- **Indices** and **value** may not be **materialized in memory**

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

Sparse Matrix - *Differences* from `std::unordered_map`

- `key_type` is **pair-like type** filled with **integral values**
- **Matrix shape** restricts **valid key values**
- **Implementation** will use **highly specialized sparse matrix formats**
- **Indices** and **value** may not be **materialized in memory**

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

Sparse Matrix - *Differences* from `std::unordered_map`

- `key_type` is **pair-like type** filled with **integral values**
- **Matrix shape** restricts **valid key values**
- **Implementation** will use **highly specialized sparse matrix formats**
- **Indices** and **value** may not be **materialized in memory**

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

Sparse Matrix - *Differences* from `std::unordered_map`

- `key_type` is **pair-like type** filled with **integral values**
- **Matrix shape** restricts **valid key values**
- **Implementation** will use **highly specialized sparse matrix formats**
- **Indices** and **value** may not be **materialized in memory**

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

Sparse Matrix - *Differences* from `std::unordered_map`

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

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


auto iter = x.begin();

[blank] value = *iter;
```

[{0, 1}] → 120

[{2, 3}] → 122

[{4, 3}] → 121

[{7, 0}] → 98

Sparse Matrix - *Differences* from `std::unordered_map`

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

unordered_map<key_type, m

auto iter = x.begin();

[blank] value = *iter;
```

What is the type of `*iter`?

[{0, 1}] → 120

[{2, 3}] → 122

[{4, 3}] → 121

[{7, 0}] → 98

Sparse Matrix - *Differences* from `std::unordered_map`

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

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


auto iter = x.begin();

[blank] value = *iter;
```

[{0, 1}] → 120

[{2, 3}] → 122

[{4, 3}] → 121

[{7, 0}] → 98

Sparse Matrix - *Differences* from `std::unordered_map`

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

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


auto iter = x.begin();

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

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

Sparse Matrix - *Differences* from `std::unordered_map`

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

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


auto iter = x.begin();

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

[{0, 1}] → 120

[{2, 3}] → 122

[{4, 3}] → 121

[{7, 0}] → 98

Sparse Matrix - *Differences* from `std::unordered_map`

```
using key_type = st
using map_type = in

unordered_map<key_t


auto iter = x.begin(

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

value_type& value = *iter;
```

1. Each element exists **materialized** somewhere

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

[{0, 1}] → 120

[{2, 3}] → 122

[{4, 3}] → 121

[{7, 0}] → 98

(Possible) Physical Memory Layout

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

Sparse Matrix - *Differences* from `std::unordered_map`

```
using key_type = st
using map_type = in

unordered_map<key_t

auto iter = x.begin(

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

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

[{0, 1}] → 120

[{2, 3}] → 122

[{4, 3}] → 121

[{7, 0}] → 98

(Possible) Physical Memory Layout

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

Sparse Matrix - *Differences* from `std::unordered_map`

```
using key_type = std::p
using map_type = int;

unord                        e,

auto iter = x.begin();

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

`value_type&`

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

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

(Possible) Physical Memory Layout

{{0, 1}, **120**}	{{4, 3}, **121**}	{{7, 0}, **98**}	{{2, 3}, **122**}

Sparse Matrix - *Differences* from `std::unordered_map`

```
using key_type = std::p
using map_type = int;

unordered_map<key_type,

auto iter = x.begin();

using                              key_type,
                              map_type>;
value_type& value = *iter;
```

1. Each element exists **materialized** somewhere

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

`const pair<int, int>&`

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

(Possible) Physical Memory Layout

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

Sparse Matrix - *Differences* from `std::unordered_map`

```
using key_type = std::p
using map_type = int;

unordered_map<key_type,

auto iter = x.begin();

using                        key_type,
                        map_type>;
value_type& value = *iter;
```

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

`int&`

`const pair<int, int>&`

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

(Possible) Physical Memory Layout

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

Sparse Matrix Formats

- Need to enable **a variety** of different **sparse matrix formats**
- Most formats **separate values** and **indices**, may not store some indices
- This means we need to use a **custom reference type** for indices

Compressed Sparse Row (CSR) Storage Format

Row Pointers

0	2	3	3	3	6	6	7

Column Indices

0	2	2	2	3	4	3

Values

8	2	5	7	1	2	9

Sparse Matrix Representation

8		2	
		5	
	7	1	2
		1	

Sparse Matrix Formats

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

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

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

Compressed Sparse Row (CSR) Storage Format

Row Pointers

0	2	3	3	3	6	6	7

Column Indices

0	2	2	2	3	4	3

Values

8	2	5	7	1	2	9

Sparse Matrix Representation

8		2	
		5	
	7	1	2
		1	

Sparse Matrix Formats

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

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

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

Compressed Sparse Row (CSR) Storage Format

Row Pointers

0	2	3	3	3	6	6	7

Column Indices

0	2	2	2	3	4	3

Values

8	2	5	7	1	2	9

Sparse Matrix Representation

8		2	
		5	
	7	1	2
		1	

Sparse Matrix Formats

- Need to enable **a variety** of different **sparse matrix formats**
- Most formats **separate values** and **indices**, may not store some indices
- This means we need to use a **custom reference type** for indices

Compressed Sparse Row (CSR) Storage Format

Row Pointers

0	2	3	3	3	6	6	7

Column Indices

0	2	2	2	3	4	3

Values

8	2	5	7	1	2	9

Sparse Matrix Representation

8		2	
		5	
	7	1	2
		1	

Sparse Matrix Formats

- Need to enable **a variety** of different **sparse matrix formats**
- Most formats **separate values** and **indices**, may not store some indices
- This means we need to use a **custom reference type** for indices

Compressed Sparse Row (CSR) Storage Format

Row Pointers

0	2	3	3	3	6	6	7

Column Indices

0	2	2	2	3	4	3

Values

8	2	5	7	1	2	9

Sparse Matrix Representation

8		2	
		5	
	7	1	2
		1	

Sparse Matrix Formats

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

- Most formats **separate val**[...] **indices**, may[...]

- This means w[...] need to use a **custom reference type** for indices

Custom reference type, like `vector<bool>::reference`

Compressed Sparse Row (CSR) Storage Format

Row Pointers

0	2	3	3	3	6	6	7

Column Indices

0	2	2	2	3	4	3

Values

8	2	5	7	1	2	9

Sparse Matrix Representation

8		2	
		5	
	7	1	2
		1	

Matrix Data Structure

```
grb::matrix<float>
```

Type of stored values

Matrix Data Structure

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

Type of stored values

(Integer) type used to store indices

Matrix Data Structure: Attributes

Attributes

Shape

Dimensions of matrix

(Graph: number of vertices)

Size

Number of stored values

(Graph: number of edges)

```
grb::matrix<float> x({1024, 1024});

size_t m = x.shape()[0];
size_t n = x.shape()[1];

size_t nnz = x.size();
```

Matrix Data Structure: Element Access

Element Access

Direct access to stored values

operator[]

Find or insert value by index

find

Find value by index

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

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

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

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

Matrix Data Structure: Iteration

Iteration

Iteration over stored values

Can read: row, column, value

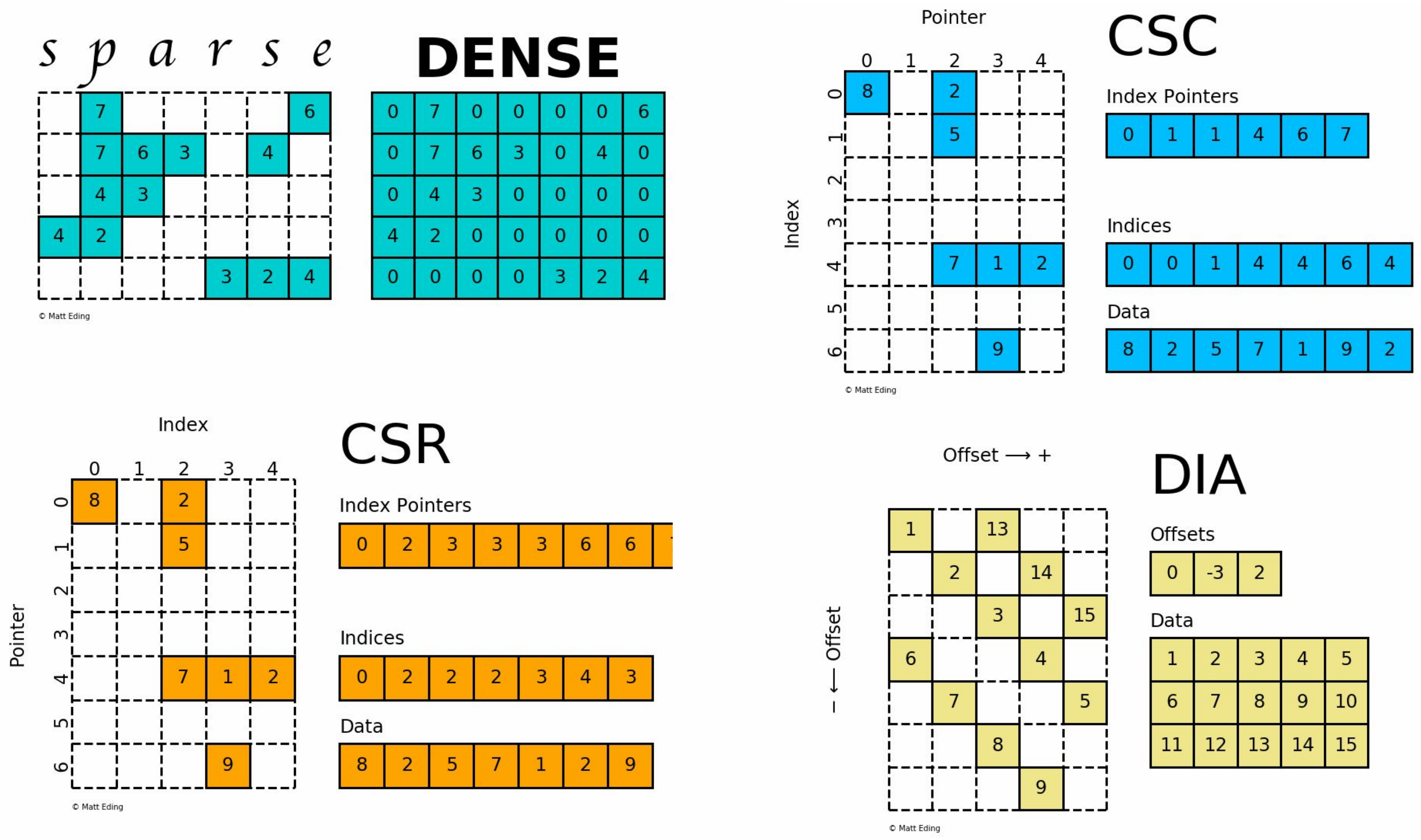
Can write: value only

Iteration allows support for standard C++ algorithms.

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

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

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

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

Sparse Matrix Formats

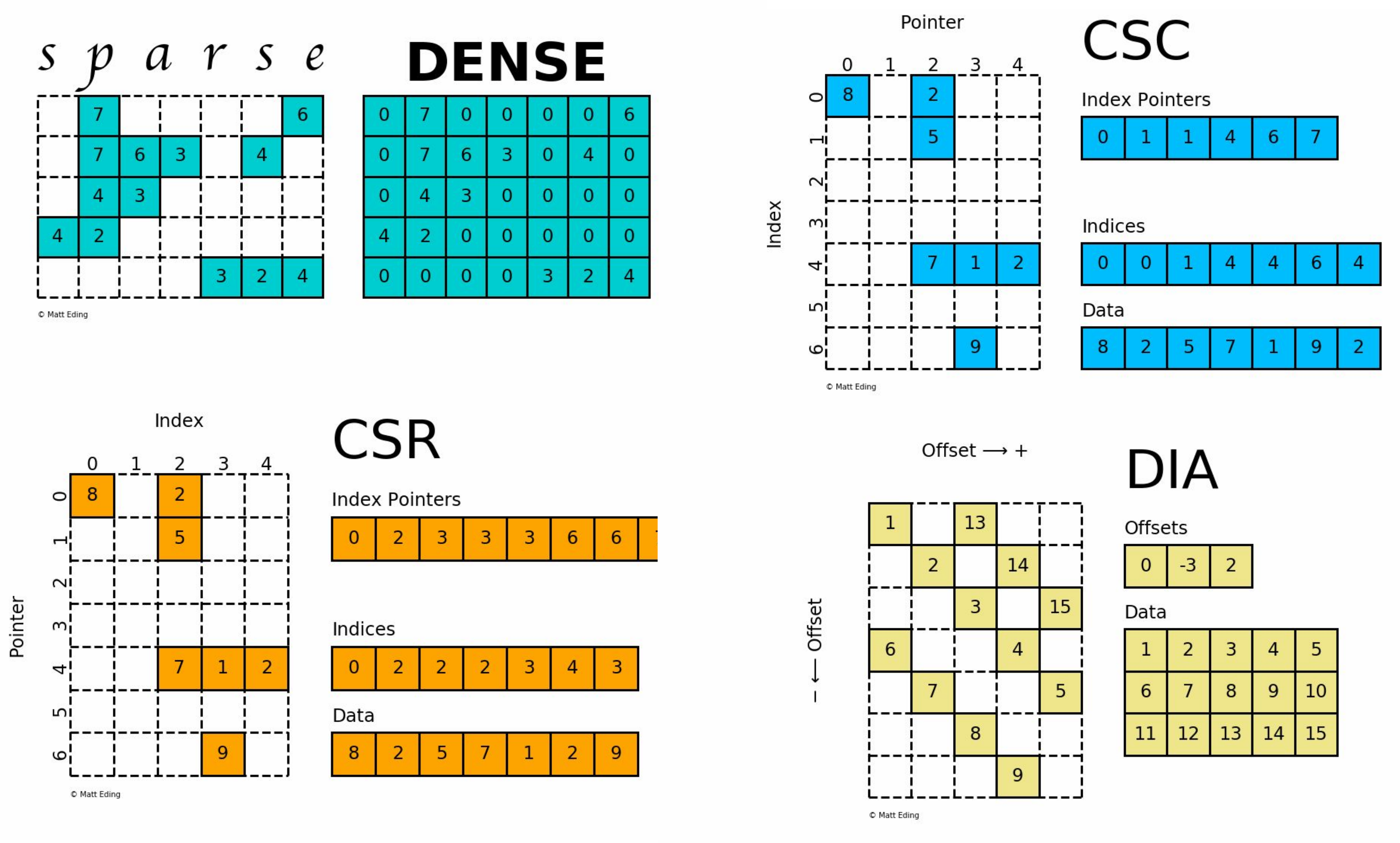
- Many potential **sparse matrix formats**
- Each format has **different iteration patterns**
- Inefficient to enforce a **particular iteration order**



* Diagrams by Matt Eding at https://matteding.github.io/2019/04/25/sparse-matrices/

Sparse Matrix Formats

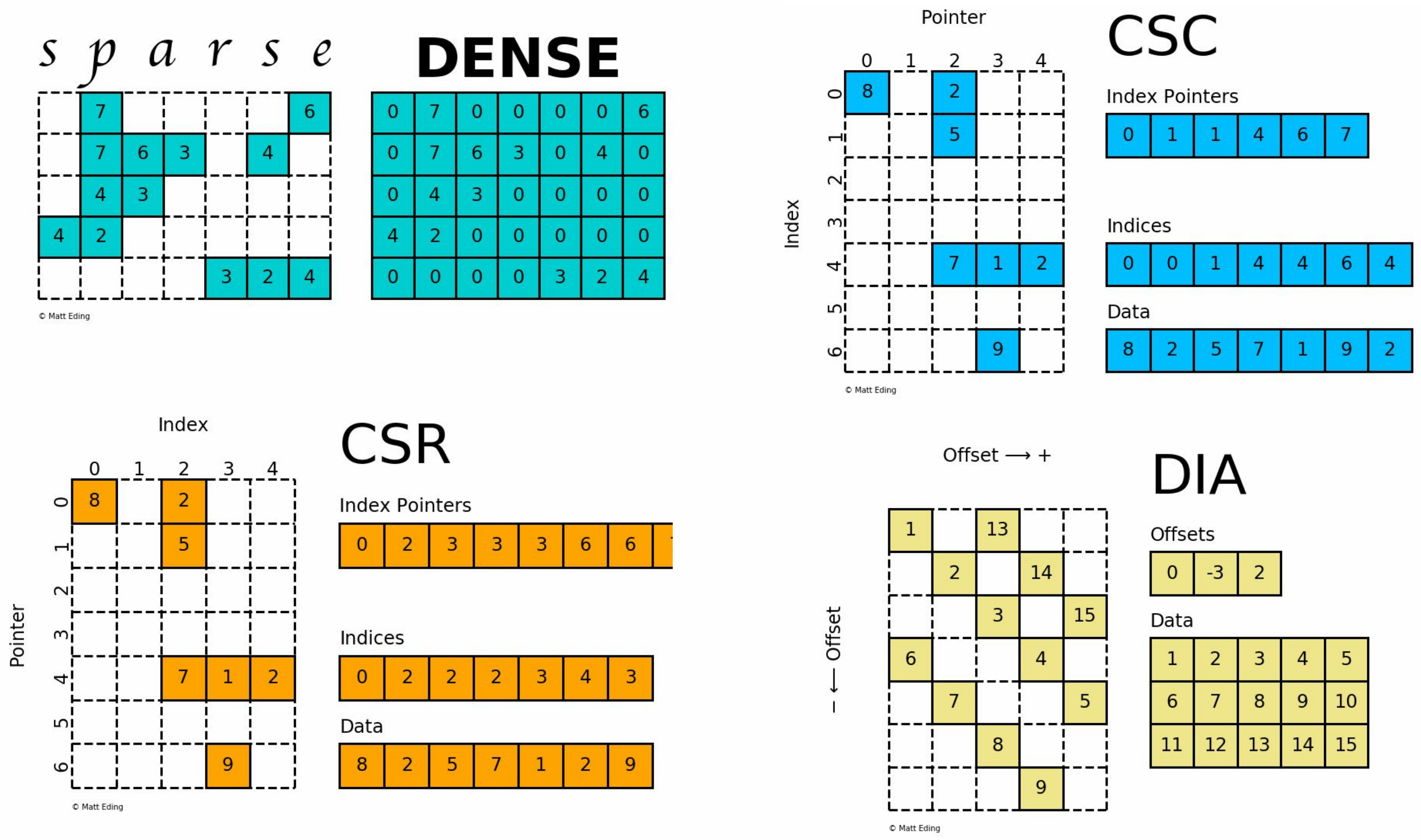
- Many potential **sparse matrix formats**
- Each format has **different iteration patterns**
- Inefficient to enforce a **particular iteration order**



* Diagrams by Matt Eding at https://matteding.github.io/2019/04/25/sparse-matrices/

Sparse Matrix Formats

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



* Diagrams by Matt Eding at https://matteding.github.io/2019/04/25/sparse-matrices/

Matrix Data Structure

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

Type of stored values

(Integer) type used to store indices

Compile-time hint about storage format

Matrix Data Structure: Iteration

- **Unordered iteration** over stored values

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

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

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

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

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

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

Matrix Data Structure: Iteration

- **Unordered iteration** over stored values
- Range of **`size()`** **`matrix_entry<T, I>`** elements
- **Tuple-like type** with access to **indices** and T& reference to value

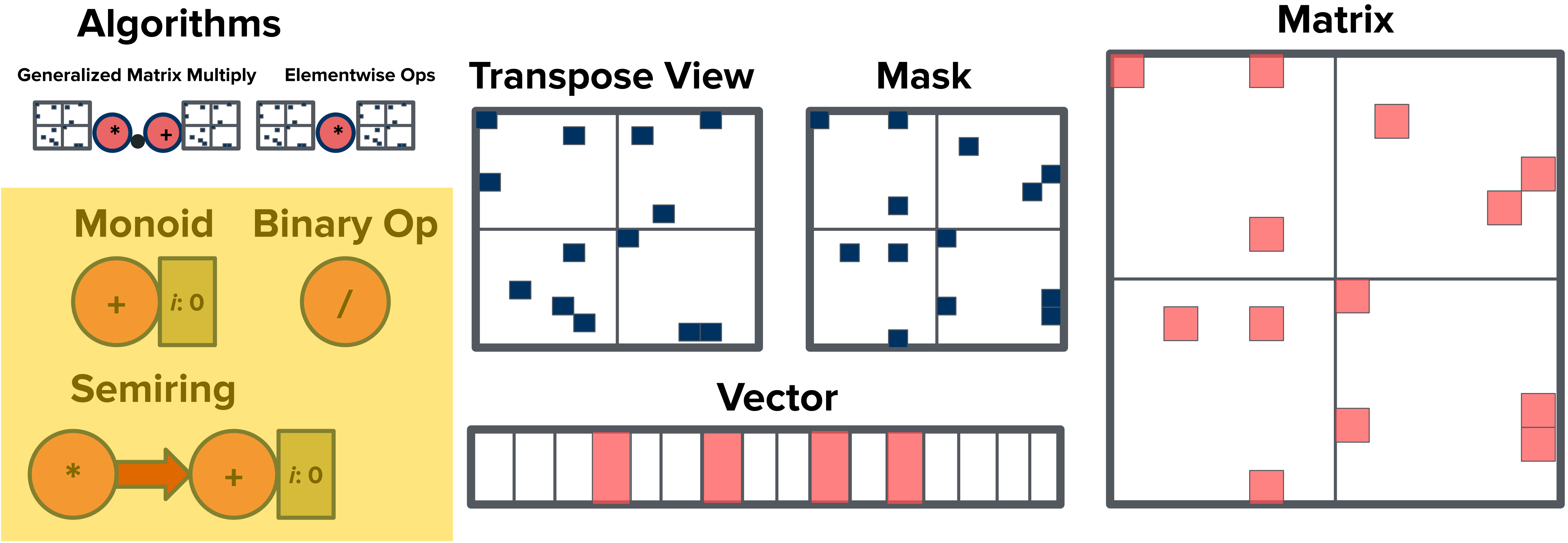
```
grb::matrix<float, int> m = ...;

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

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

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

Matrix Data Structure: Iteration

- **Unordered iteration** over stored values
- Range of **`size()`** **`matrix_entry<T, I>`** elements
- **Tuple-like type** with access to **indices** and T& reference to value

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

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

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

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

GraphBLAS Concepts



Binary Operators

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

`T x U -> V`

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

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

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

grb::ewise_mult(c, ..., a, b, my_op);
```

Monoids: Binary Operators with an Identity

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

- A **commutative** binary operator

- A type **T**

- A mathematical **identity**

Monoids: Binary Operators with an Identity

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

- A **commutative** binary operator

- A type **T**

- A mathematical **identity**

Monoids: Binary Operators with an Identity

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

- A **commutative** binary operator

- A type **T**

- A mathematical **identity**

Monoids: Binary Operators with an Identity

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

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

Monoids

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

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

- Depends on whether **monoid_traits** specialization exists

```
using grb;

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

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

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

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

Monoids

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

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

- Depends on whether **monoid_traits** specialization exists

```
using grb;

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

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

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

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

Carnegie Mellon University
Software Engineering Institute

Berkeley
UNIVERSITY OF CALIFORNIA

Monoids

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

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

- Depends on whether **monoid_traits** specialization exists

```
using grb;

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

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

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

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

Monoids

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

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

- Depends on whether **monoid_traits** specialization exists

```
using grb;

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

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

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

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

Obtaining a Monoid

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

```
using grb;

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

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

std::cout << g << " " << s << std::endl;
```

Obtaining a Monoid

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

```
using grb;

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

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

std::cout << g << " " << s << std::endl;
```

Obtaining a Monoid

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

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

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

- Use **`make_monoid`** helper function

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

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

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

Obtaining a Monoid

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

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

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

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

Obtaining a Monoid

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

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

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

Semirings

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

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

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

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

auto my_times = [](auto a, auto b) {
                  return a*b;
                };
auto my_plus = [](auto a, auto b) {
                 return a+b;
               };
auto m_plus = grb::make_monoid(my_plus, 0);

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

Semirings

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

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

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

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

auto my_times = [](auto a, auto b) {
                  return a*b;
                };
auto my_plus = [](auto a, auto b) {
                 return a+b;
               };
auto m_plus = grb::make_monoid(my_plus, 0);

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

Semirings

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

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

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

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

auto my_times = [](auto a, auto b) {
                  return a*b;
                };
auto my_plus = [](auto a, auto b) {
                 return a+b;
               };
auto m_plus = grb::make_monoid(my_plus, 0);

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

Semirings

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

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

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

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

auto my_times = [](auto a, auto b) {
                  return a*b;
                };
auto my_plus = [](auto a, auto b) {
                 return a+b;
               };
auto m_plus = grb::make_monoid(my_plus, 0);

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

Semirings

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

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

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

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

auto my_times = [](auto a, auto b) {
                  return a*b;
                };
auto my_plus = [](auto a, auto b) {
                 return a+b;
               };
auto m_plus = grb::make_monoid(my_plus, 0);

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

Semirings

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

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

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

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

auto my_times = [](auto a, auto b) {
                  return a*b;
                };
auto my_plus = [](auto a, auto b) {
                 return a+b;
               };
auto m_plus = grb::make_monoid(my_plus, 0);

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

Semirings

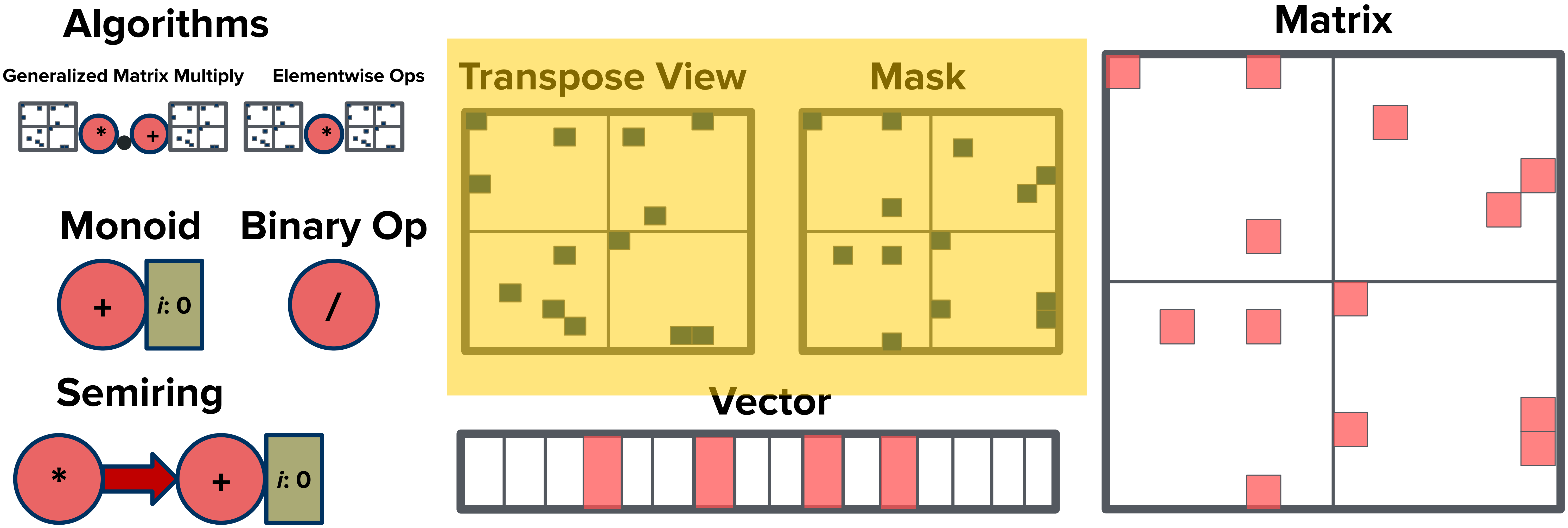
Semirings combine a **binary op b** and a **monoid m**, where **b** distributes over **m**

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

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

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

auto my_times = [](auto a, auto b) {
                   return a*b;
                 };
auto my_plus = [](auto a, auto b) {
                  return a+b;
                };
auto m_plus = grb::make_monoid(my_plus, 0);

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

GraphBLAS Concepts



Views

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

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

This simplifies API, removes some of need for **descriptors**.

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

auto a_t = grb::transpose(a);

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

Views

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

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

This simplifies API, removes some of need for **descriptors**.

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

auto a_t = grb::transpose(a);

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

Matrix Transform Views

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

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

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

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

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

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

Matrix Transform Views

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

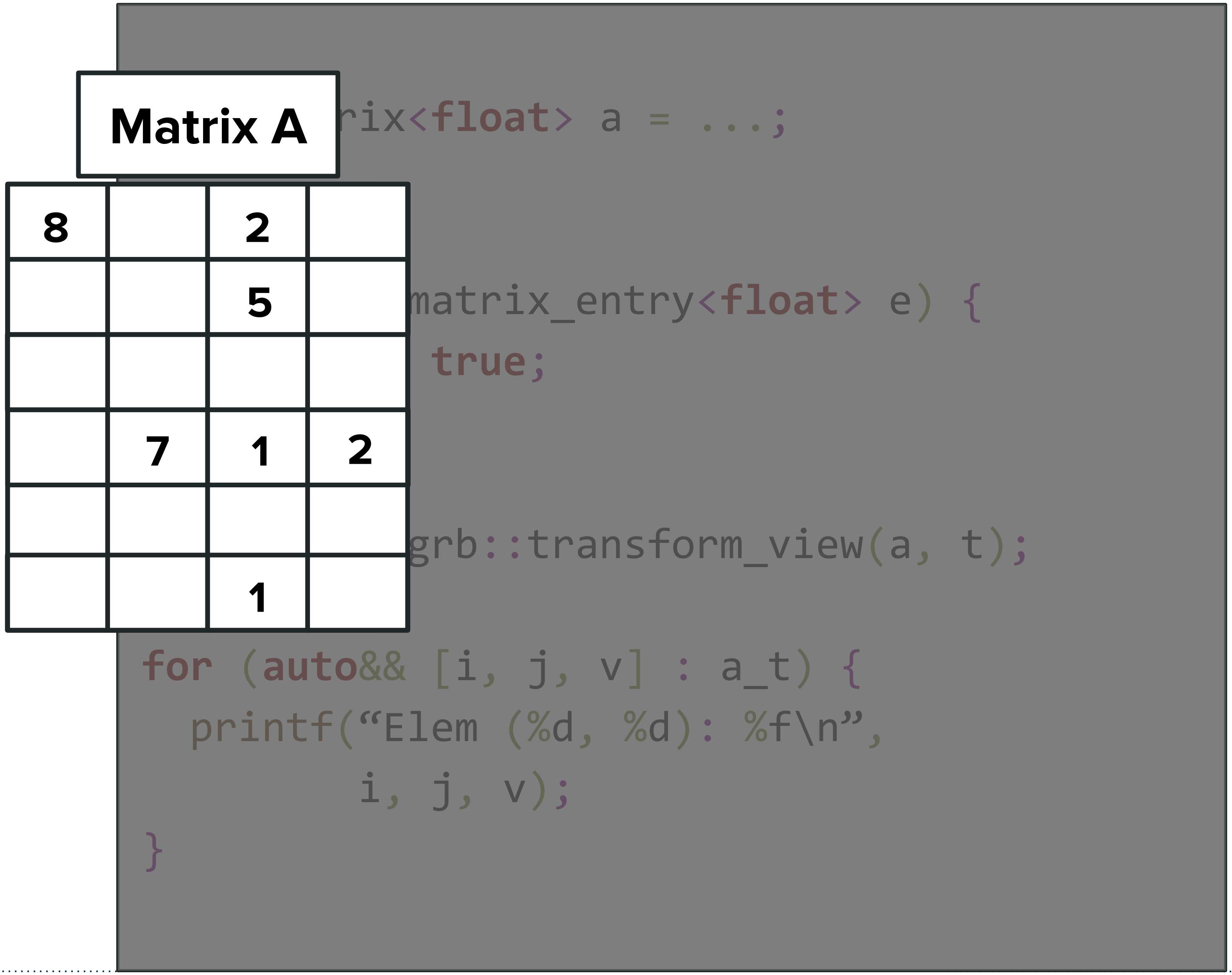
- Can be used to create **structure-only view**

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

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

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

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

Matrix Transform Views

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

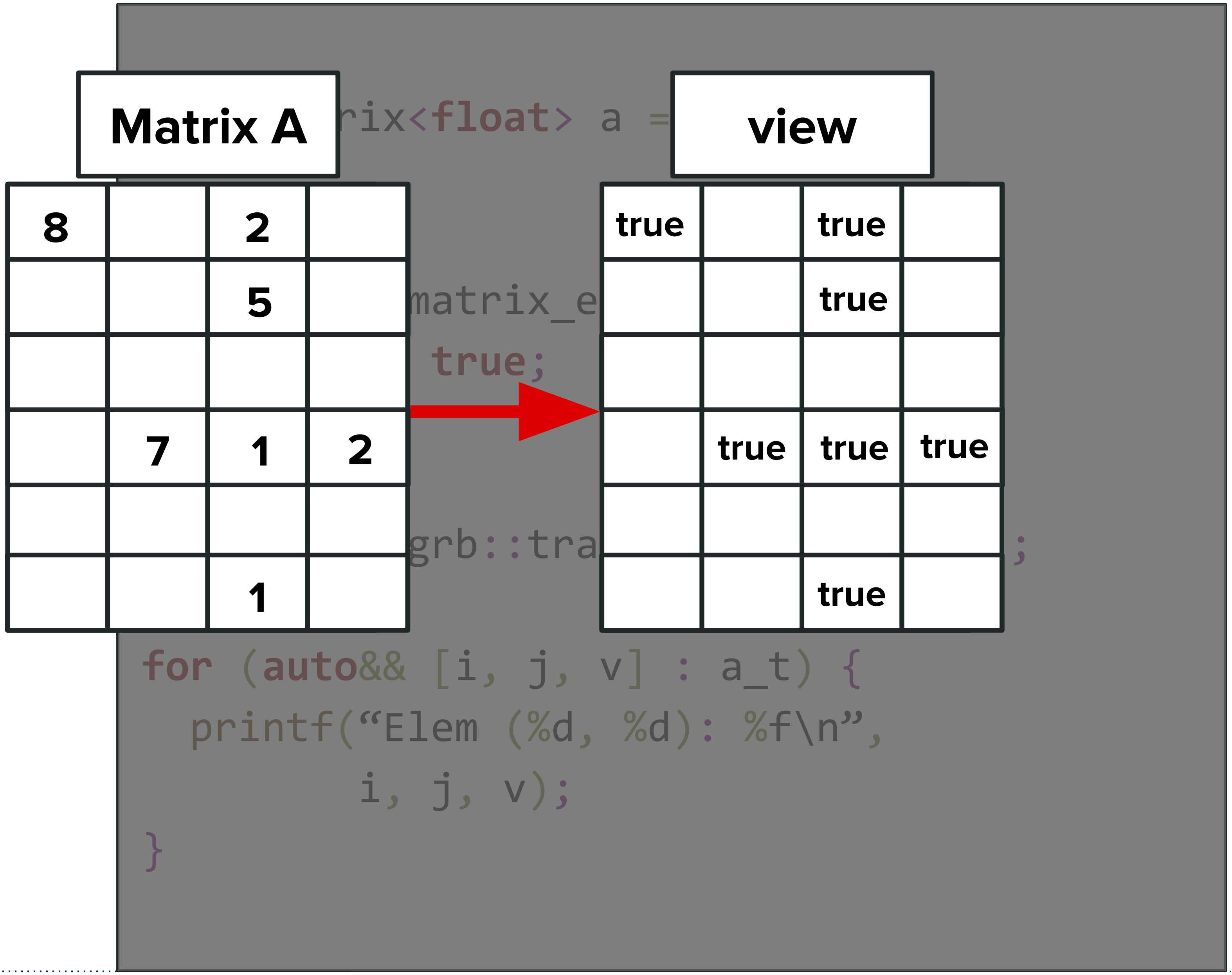
- Can be used to create **structure-only view**



Matrix Transform Views

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

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



GraphBLAS Masks

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

$$
\begin{bmatrix} 8 & & 2 & \\ & & 5 & \\ & & & \\ & 7 & 1 & 2 \end{bmatrix}
\times
\begin{bmatrix} 1 \\ \\ 1 \\ 1 \end{bmatrix}
=
\begin{bmatrix} 10 \\ 5 \\ \\ 3 \end{bmatrix}
$$

GraphBLAS Masks

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

$$\begin{bmatrix} 8 & & 2 & \\ & & 5 & \\ & & & \\ & 7 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} 1 \\ \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ \\ 3 \end{bmatrix}$$

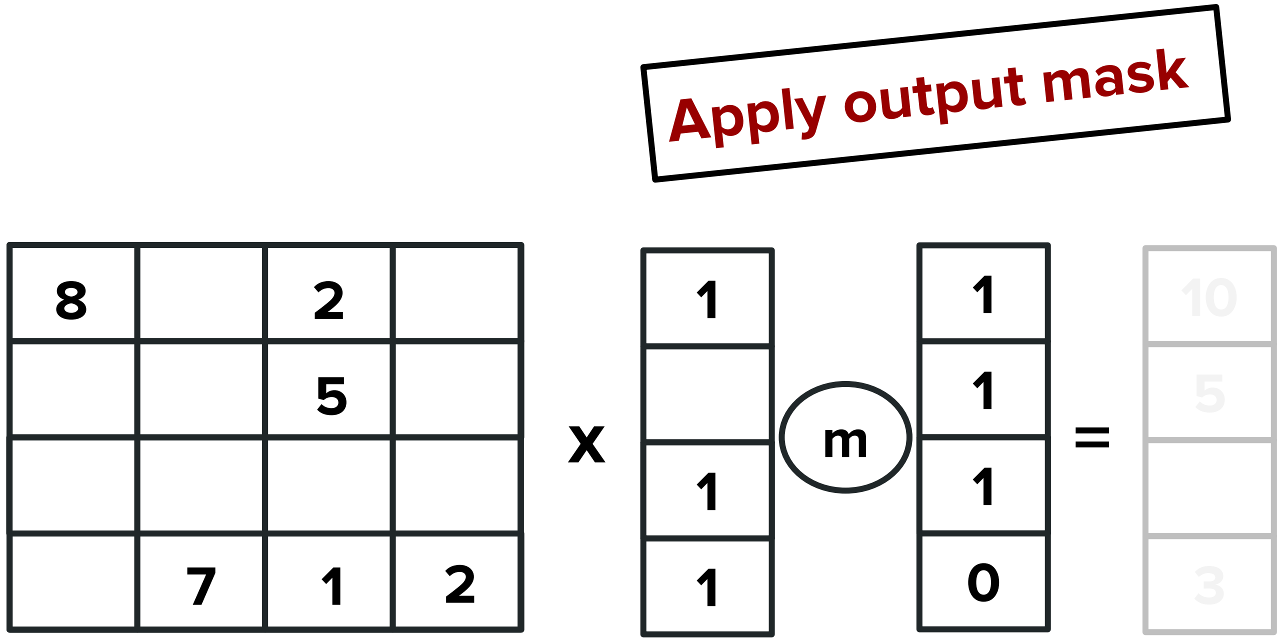
GraphBLAS Masks

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

$$\begin{bmatrix} 8 & & 2 & \\ & & 5 & \\ & & & \\ & 7 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} 1 \\ \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ \\ 3 \end{bmatrix}$$

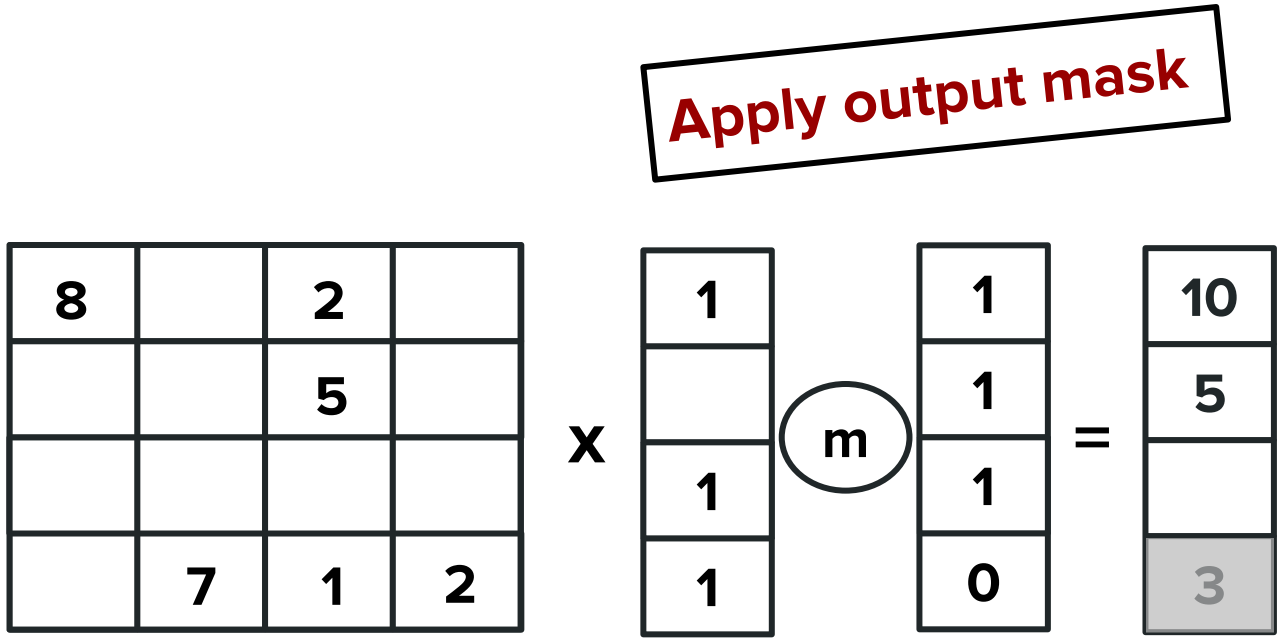
GraphBLAS Masks

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



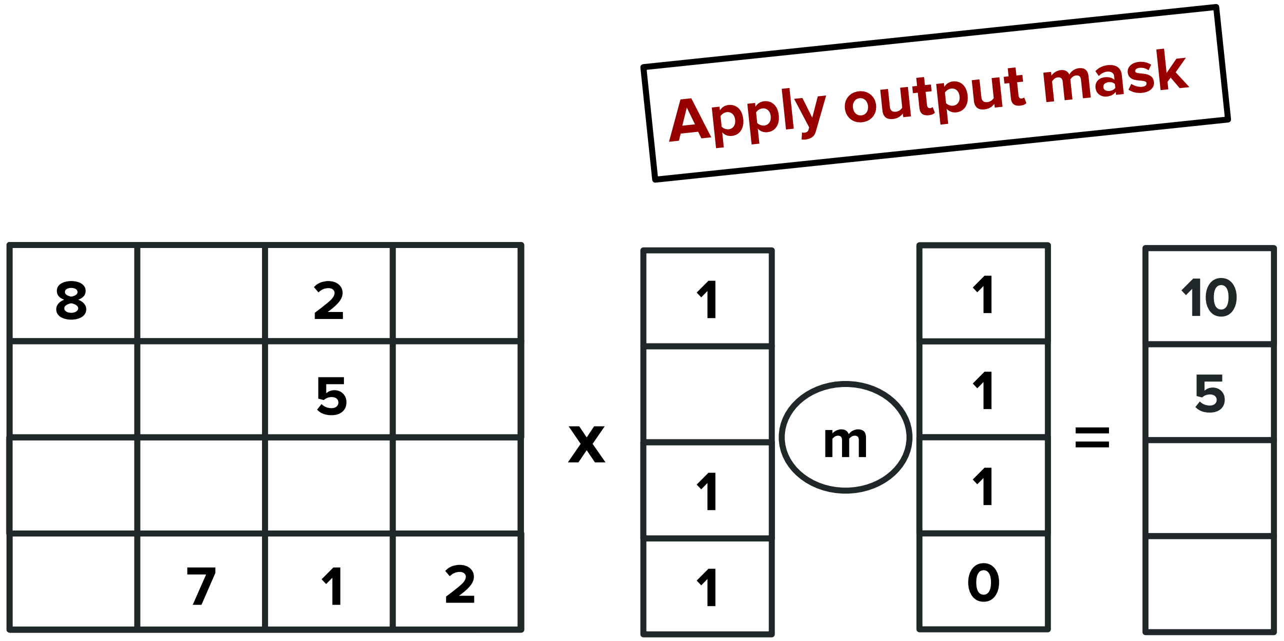
GraphBLAS Masks

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



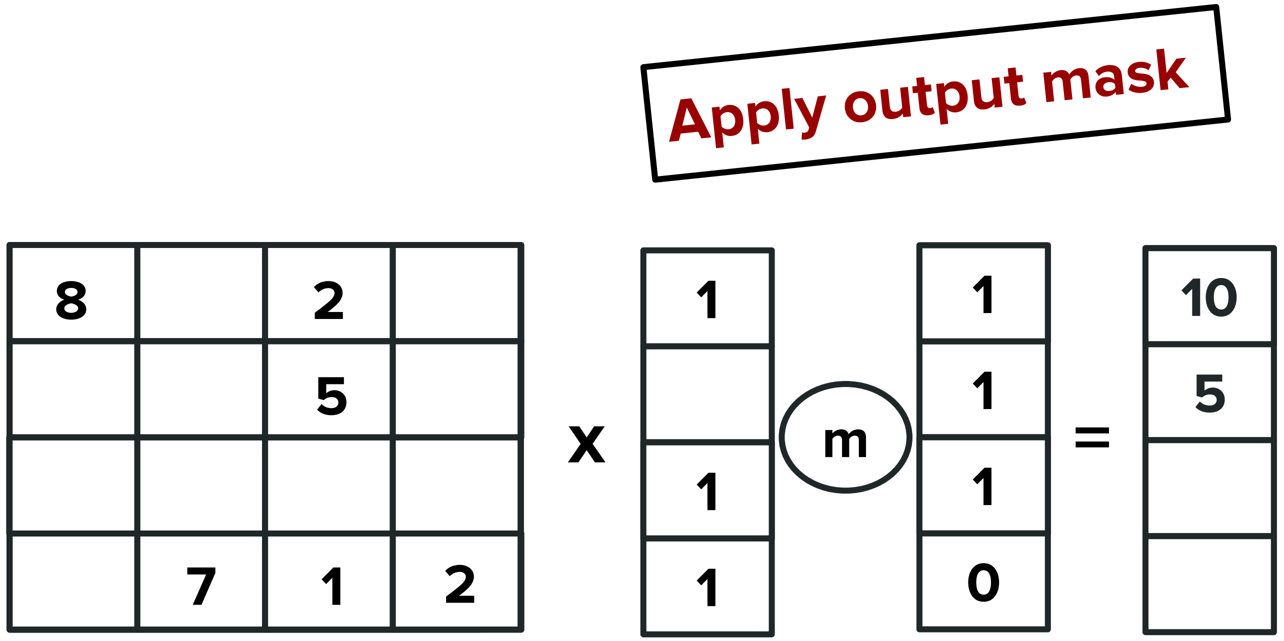
GraphBLAS Masks

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



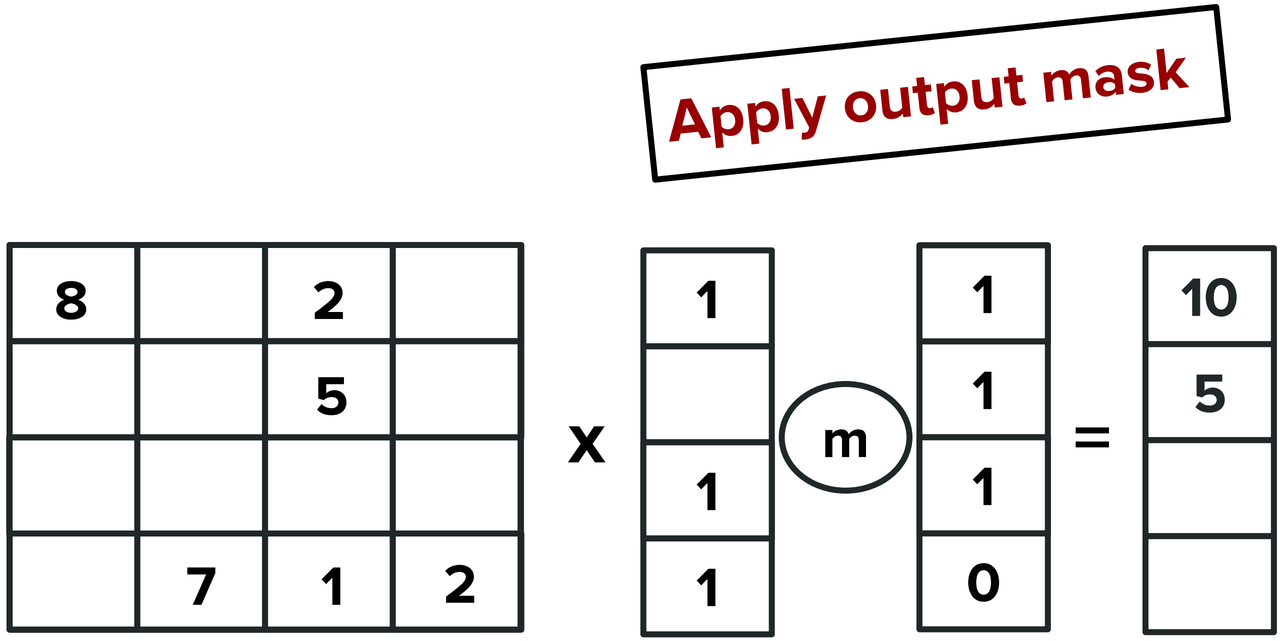
GraphBLAS Masks

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



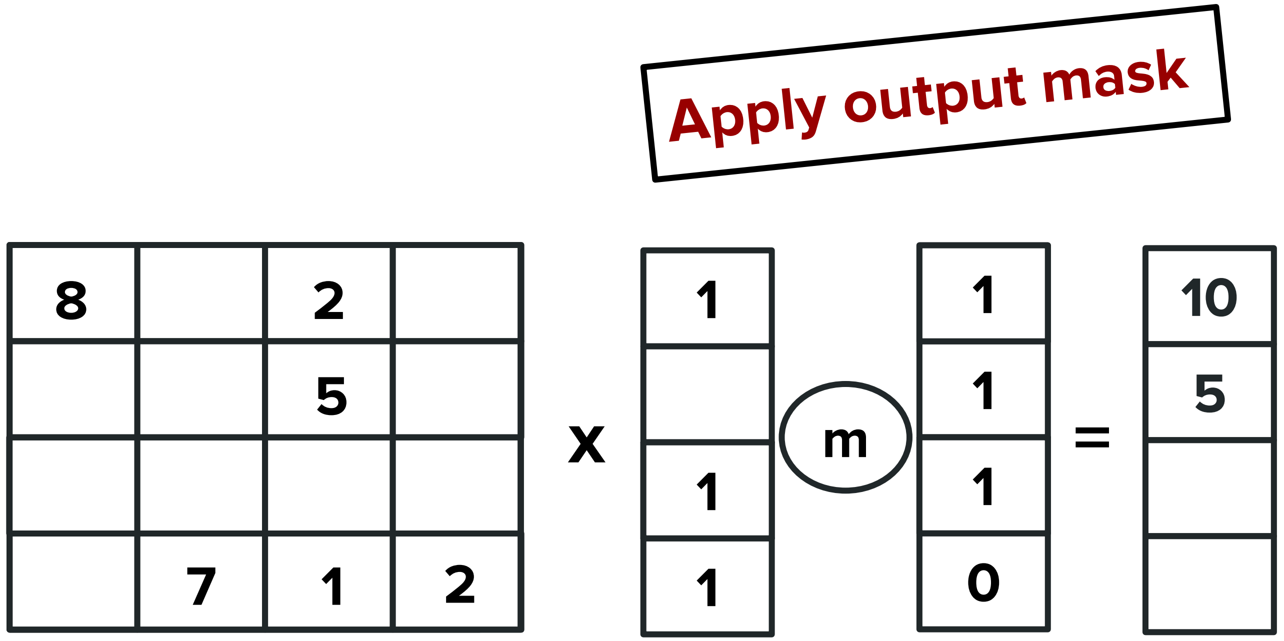
GraphBLAS Masks

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



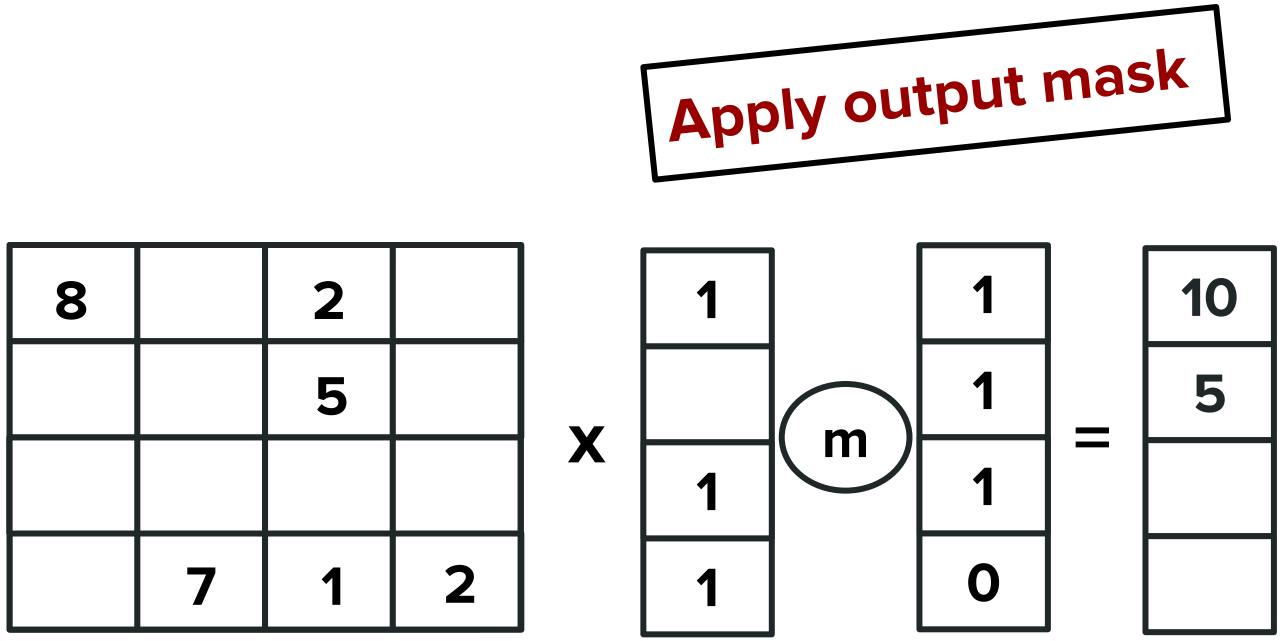
GraphBLAS Masks

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

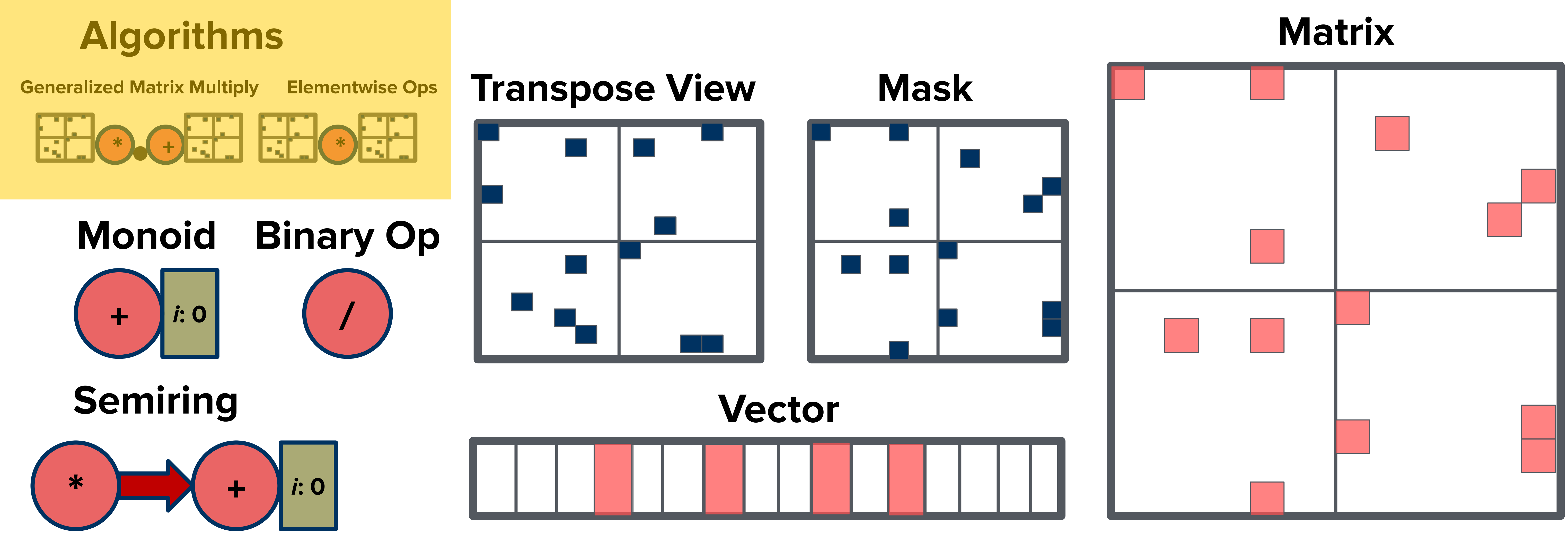


GraphBLAS Masks

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



GraphBLAS Concepts



Algorithms

The primary algorithms of interest are:

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

2) **Elementwise** operations

Matrix Multiply

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

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

Similar to C API

```
using grb;

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

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

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

Matrix Multiply

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

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

Similar to C API

```
using grb;

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

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

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

Matrix Multiply

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

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

Similar to C API

```
using grb;

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

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

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

Matrix Multiply

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

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

Similar to C API

```
using grb;

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

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

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

Matrix Multiply

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

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

Similar to C API

```
using grb;

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

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

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

Matrix Multiply

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

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

Similar to C API

```
using grb;

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

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

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

Matrix Multiply

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

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

Similar to C API

```
using grb;

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

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

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

Matrix Multiply Definition

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

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

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

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

Matrix Multiply Definition

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

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

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

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

Matrix Multiply Definition

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

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

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

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

Matrix Multiply Definition

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

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

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

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

Matrix Multiply Definition

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

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

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

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

Matrix Multiply Definition

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

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

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

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

Matrix Multiply Definition

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

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

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

```
template <MatrixRange C,
          typename Accumulator,
          ConstMatrixRange A,
          ConstMatrixRange B,
          MaskMatrixRange M,
          typename Semiring>
void mxm(C&& c, Accumulator&& acc,
         A&& a, B&& b,
         M&& mask, Semiring&& s);
```

Matrix Times Matrix

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

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

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

Matrix Times Matrix (mxm)

Very **similar to C API**

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

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

Interoperability with C++ Algorithms

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

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

- **One dimensional iteration** somewhat limited, but **2D iteration** concepts are coming (next slide)

Interoperability with C++ Algorithms

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

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

- **One dimensional iteration** somewhat limited, but **2D iteration** concepts are coming (next slide)

Interoperability with C++ Algorithms

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

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

- **One dimensional iteration** somewhat limited, but **2D iteration** concepts are coming (next slide)

Interoperability with C++ Graph Library

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

- We aren't currently using **multidimensional iteration**

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

Wrap-Up

We can use **matrix algebra** to implement graph algorithms

Can support a variety of different **sparse matrix formats**

Provide **high-level interfaces** for algorithms

Brief Advertisement

If you enjoy **parallel programming**:

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

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

Virtual: **Wednesday, 12:30 PM MDT**

GraphBLAS Links

graphblas.org

github.com/cmu-sei/gbtl

github.com/BenBrock/rgri

Ben Brock, PhD Candidate at **UC Berkeley**

Data structures and algorithms for large-scale parallel systems.

Please hire me!

Scott McMillan, Principal Research Engineer at **CMU SEI**

Graph/ML/AI algorithms for large- and small-scale parallel systems.