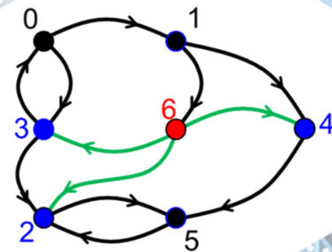


The GraphBLAS C++ API:
C++ and Interoperability Between Libraries

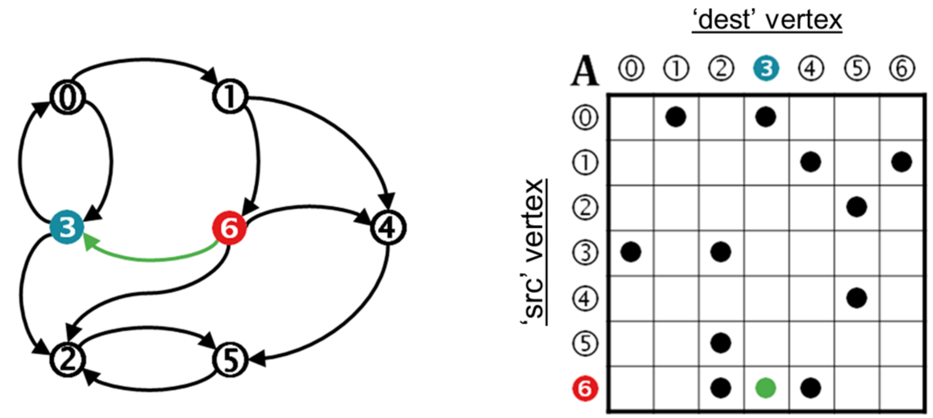
Benjamin Brock, *Intel*
Scott McMillan, *CMU Software Engineering Institute*
Aydin Buluc, *Lawrence Berkeley National Laboratory*
Jose E. Moreira, *IBM Research*
Timothy G. Mattson, *Intel*

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

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.

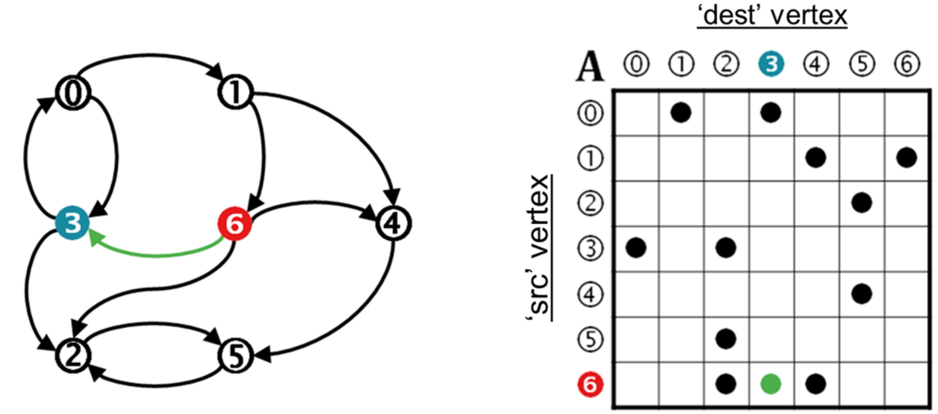
DM23-0476

GraphBLAS C API



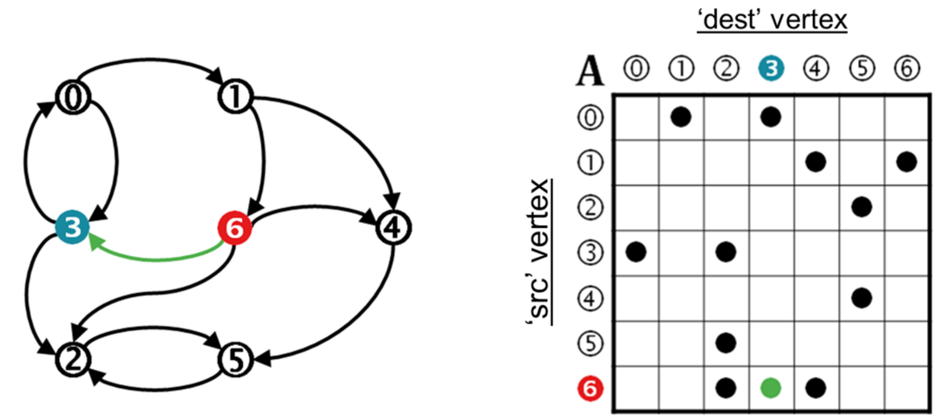
- Provides **uniform API** for **graph algorithms** in the **language of linear algebra**
- Revolve around sparse matrix and vector operations which can use **arbitrary semirings** instead of classical (+, *)
- Current version of C API Specification is 2.0
- C offers great **portability** (Python, bindings,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

GraphBLAS C API



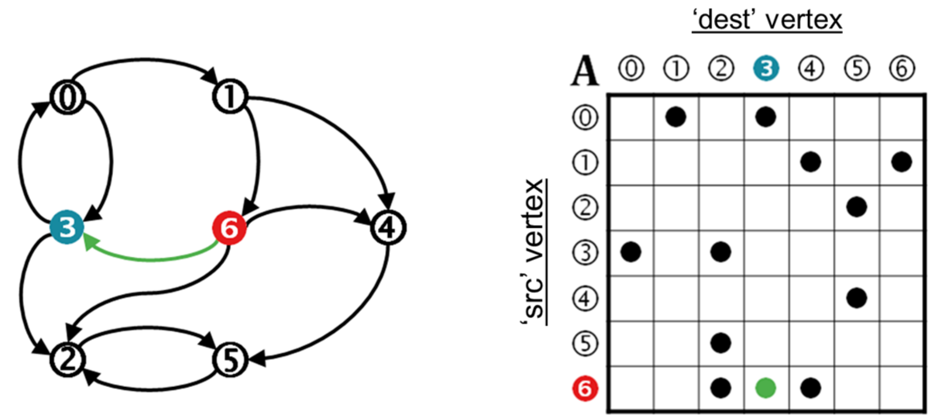
- Provides **uniform API** for **graph algorithms** in the **language of linear algebra**
- Revolve around sparse matrix and vector operations which can use **arbitrary semirings** instead of classical (+, *)
- Current version of C API Specification is 2.0
- C offers great **portability** (Python, bindings,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

GraphBLAS C API



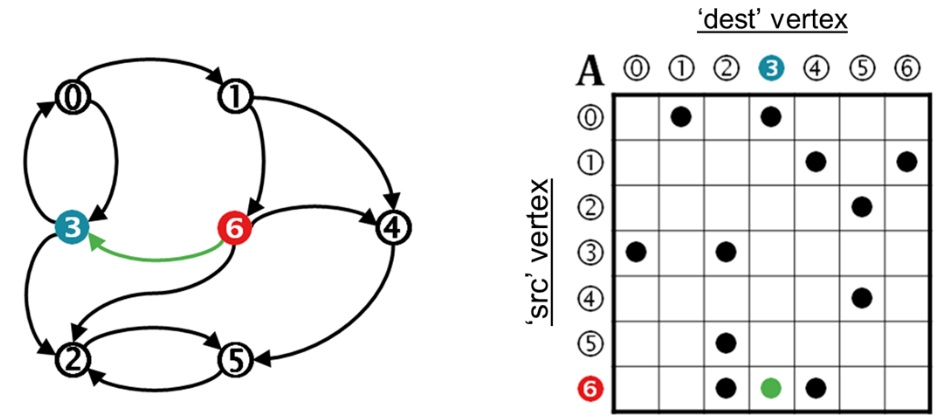
- Provides **uniform API** for **graph algorithms** in the **language of linear algebra**
- Revolve around sparse matrix and vector operations which can use **arbitrary semirings** instead of classical (+, *)
- Current version of C API Specification is 2.0
- C offers great **portability** (Python, bindings,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

GraphBLAS C API



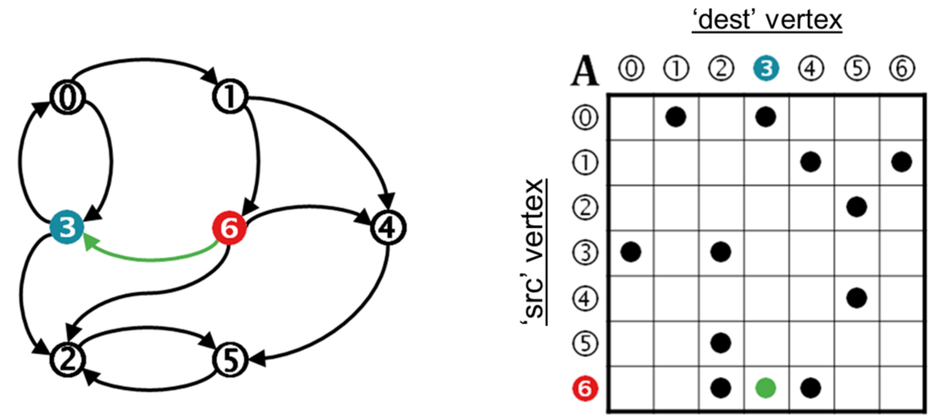
- Provides **uniform API** for **graph algorithms** in the **language of linear algebra**
- Revolve around sparse matrix and vector operations which can use **arbitrary semirings** instead of classical (+, *)
- Current version of C API Specification is 2.0
- C offers great **portability** (Python, bindings,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

GraphBLAS C API



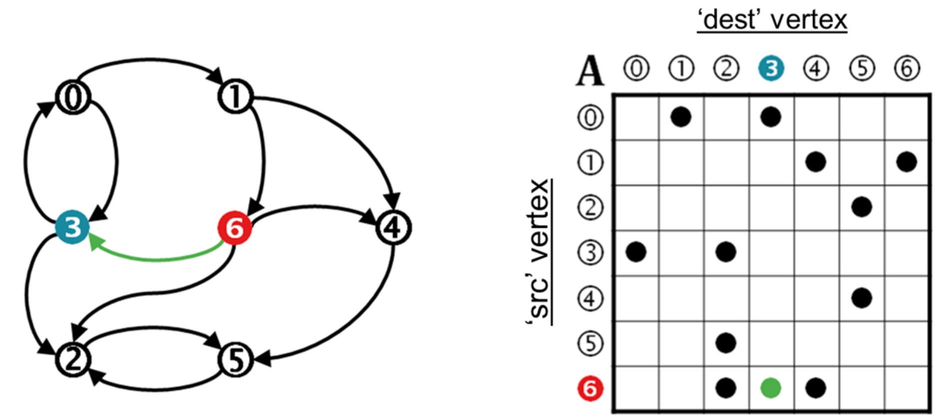
- **Generics** make C implementations **complex**
- No **introspection**, hints (e.g., types, storage, performance)
- **Interoperability** is/was not high enough priority

GraphBLAS C API



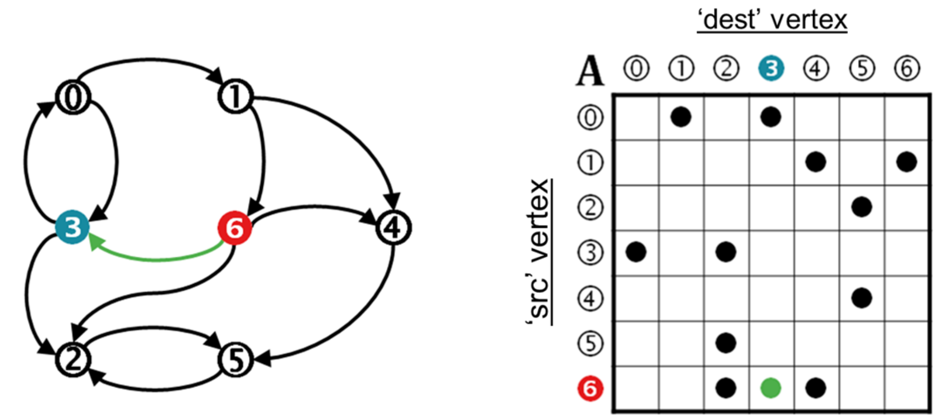
- **Generics** make C implementations **complex**
- No **introspection**, hints (e.g., types, storage, performance)
- **Interoperability** is/was not high enough priority

GraphBLAS C API



- **Generics** make C implementations **complex**
- No **introspection**, hints (e.g., types, storage, performance)
- **Interoperability** is/was not high enough priority

GraphBLAS C API



- **Generics** make C implementations **complex**
- No **introspection**, hints (e.g., types, storage, performance)
- **Interoperability** is/was not high enough priority -- John Gilbert, HPEC 2022
  - Too hard to mix GraphBLAS calls with calls to other libraries.
  - Too hard to use GraphBLAS with user data structures and code in existing packages.

Getting data in…

- Data is duplicated internally
- Complexity of **import** function not guaranteed

```
/* Multiply a matrix */
GrB_Matrix multiply(my_matrix_type* a, GrB_Matrix b)
{
    GrB_Index *rowptr = a->rowptr;
    GrB_Index *colind = a->colind;
    float     *values = a->values;

    GrB_Index   nrows = a->nrows;
    GrB_Index   ncols = a->ncols;
    GrB_Index   nvals = a->nvals;

    /* copy the data into GraphBLAS */
    GrB_Matrix grb_a;
    GrB_Matrix_import(&grb_a, GrB_FP32
                      nrows, ncols,
                      rowptr, colind, values,
                      nrows+1, nvals,  nvals,
                      GrB_CSR_FORMAT);

    GrB_Matrix c;
    GrB_mxm(c, NULL, NULL, semiring, grb_a, b, NULL);

    return c;
}
```

Getting data in…

- Data is duplicated internally
- Complexity of **import** function not guaranteed

NOTE: this can be the costliest step of an application.

```c
/* Multiply a matrix */
GrB_Matrix multiply(my_matrix_type* a, GrB_Matrix b)
{
    GrB_Index *rowptr = a->rowptr;
    GrB_Index *colind = a->colind;
    float     *values = a->values;

    GrB_Index   nrows = a->nrows;
    GrB_Index   ncols = a->ncols;
    GrB_Index   nvals = a->nvals;

    /* copy the data into GraphBLAS */
    GrB_Matrix grb_a;
    GrB_Matrix_import(&grb_a, GrB_FP32
                      nrows, ncols,
                      rowptr, colind, values,
                      nrows+1, nvals,  nvals,
                      GrB_CSR_FORMAT);

    GrB_Matrix c;
    GrB_mxm(c, NULL, NULL, semiring, grb_a, b, NULL);

    return c;
}
```

Getting data in…**and out**

- Data is duplicated externally
- Complexity of **export** function not guaranteed
- Lack of type introspection

NOTE: these issues also addressed by the C++ API but is not the focus of this presentation.

```
/* Add all the elements in a matrix */
<?type?> sumreduce(GrB_Matrix A)
{
    /* Allocate buffers for export */
    GrB_Index n_rowptr, n_colind, n_vals;
    GrB_Matrix_exportSize(&n_rowptr, &n_colidx,
                          &n_vals
                          GrB_CSR_FORMAT, A);

    GrB_Index *rowptr = /* allocate [n_rowptr] */;
    GrB_Index *colidx = /* allocate [n_colidx] */;
    <?type?>  *values = /* allocate [n_vals]   */;

    /* copy the data out */
    GrB_Matrix_export(&rowptr, &colidx, &values,
                      &n_rowptr, &n_colidx, &n_vals,
                      GrB_CSR_FORMAT, A);

    <?type?> val = 0;
    for (GrB_Index ix = 0; ix < n_vals; ++ix) {
        val += values[ix];
    }
    /* ...free memory... */

    return val;
}
```

C++ API Design Goal: Lightweight Views

- We can use **views** to allow external data structure to be used inside GraphBLAS

- A view changes the API to expose the C++ **GraphBLAS matrix concept**

```
int*   row_ptr = ...;
int*   col_ind = ...;
float*  values = ...;

auto a_view = grb::csr_matrix_view(values,
                                   rowptr,
                                   colind,
                                   m, n, nnz);

auto c = grb::multiply(a_view, b);
```

C++ API Design Goal: Lightweight Views

- We can use **views** to allow external data structure to be used inside GraphBLAS
- A view changes the API to expose the C++ **GraphBLAS matrix concept**
- This **avoids a copy**

```
int*   row_ptr = ...;
int*   col_ind = ...;
float*  values = ...;

auto a_view = grb::csr_matrix_view(values,
                                   rowptr,
                                   colind,
                                   m, n, nnz);

auto c = grb::multiply(a_view, b);
```

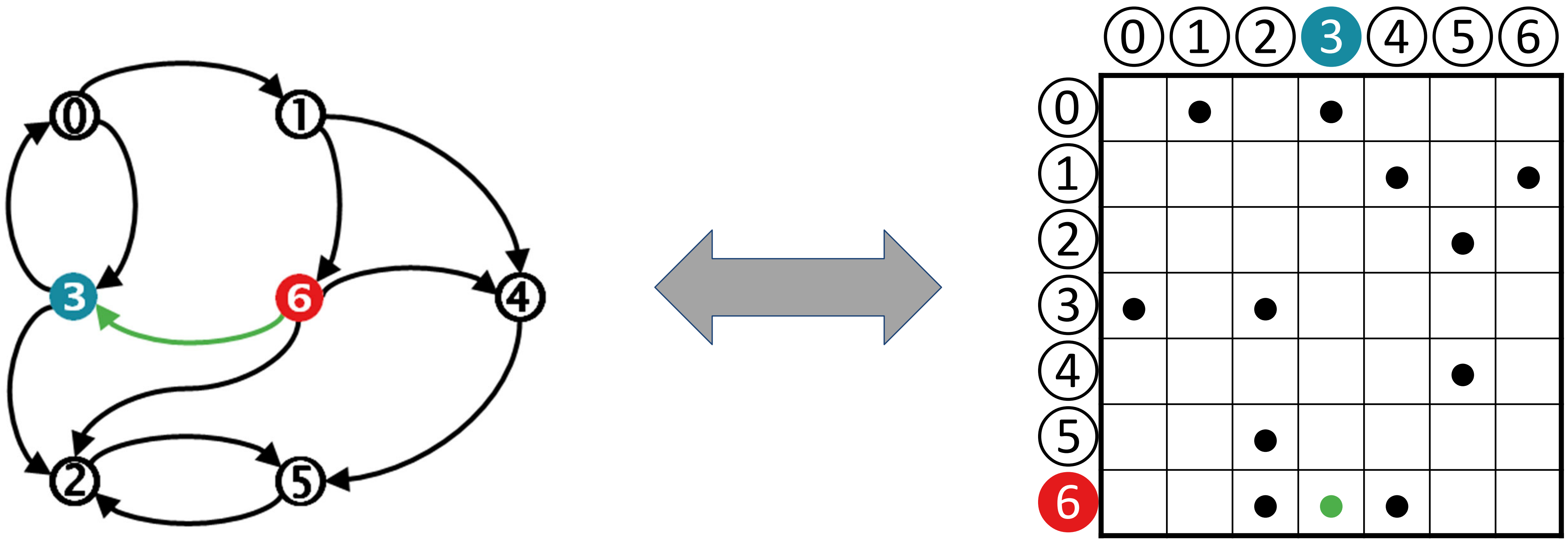
Lazy view, no copies!

Background

- C++ Concepts
- Views

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Graphs as Adjacency Matrices



Goal: define concepts that

- correspond to **sparse matrices.**
- work “like” other C++ Standard Library containers.

C++ Concepts

- Concepts describe an **interface**
- Any type that satisfies that interface **fulfills the concept**
- **Functions written in terms of concepts:** any type (M) that fulfills the concept can be passed in

```
template <grb::MatrixRange M>
grb::matrix_scalar_t<M> sumreduce(M&& A)
{
    grb::matrix_scalar_t<M> val = 0;

    for (auto&& [location, v] : A)
    {
        val += v;
    }

    return val;
}
```

C++ Concepts

- Concepts describe an **interface**
- Any type that satisfies that interface **fulfills the concept**
- **Functions written in terms of concepts:** any type (M) that fulfills the concept can be passed in

```cpp
template <grb::MatrixRange M>
grb::matrix_scalar_t<M> sumreduce(M&& A)
{
    grb::matrix_scalar_t<M> val = 0;

    for (auto&& [location, v] : A)
    {
        val += v;
    }

    return val;
}
```

C++ Concepts

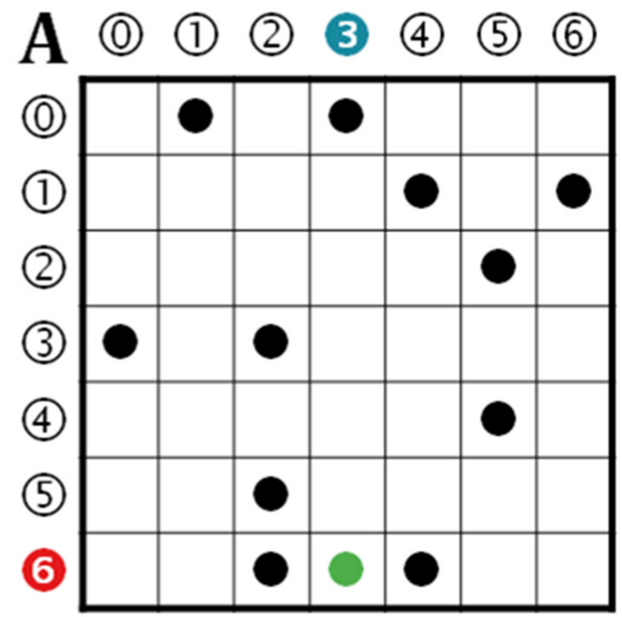
- Concepts describe an **interface**
- Any type that satisfies that interface **fulfills the concept**
- **Functions written in terms of concepts:** any type that fulfills the concept can be passed in

```
template <grb::MatrixRange M>
grb::matrix_scalar_t<M> sumreduce(M&& A)
{
    grb::matrix_scalar_t<M> val = 0;

    for (auto&& [location, v] : A)
    {
        val += v;
    }

    return val;
}
```

GraphBLAS MatrixRange Specification



- Type introspection:
  - **grb::matrix_scalar_t<M>** type of elements
  - **grb::matrix_index_t<M>** type of indices

- **shape()** - extents of the dimensions as index tuple, e.g., {7, 7}

- **size()** - number of stored elements, e.g., 12

- **find({row, col})** - access an existing value

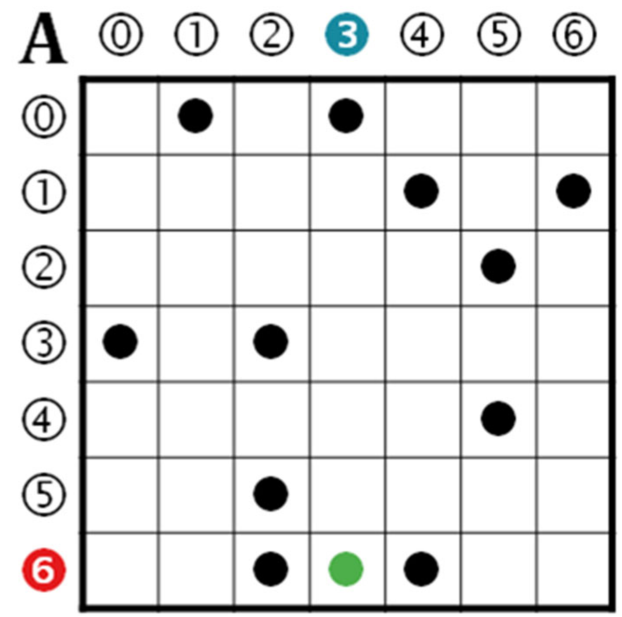
- **Forward Range:** specifies <u>unordered</u> iteration over the stored values *(illustrated on next slide)*

Aside: mutating functions like **insert()** or **erase()** are part of a refinement of MatrixRange called MutableMatrixRange.

GraphBLAS MatrixRange Example

Iteration commonly written as a **range-based for loop**:

```
template <grb::MatrixRange M>
void output_entries(M&& A) {
  for (auto&& [location, value] : A) {
    auto&& [i, j] = location;
    cout << i << ", " << j << ": " << value << endl;
  }
}
```



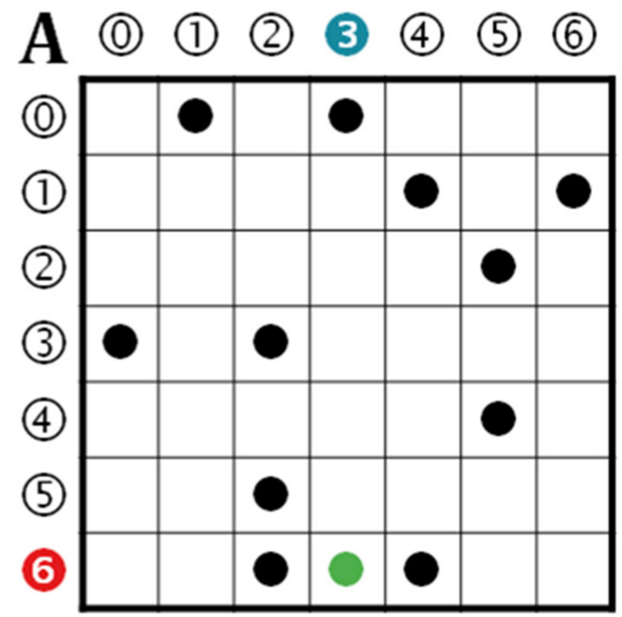
Possible output:

```
0, 1: ●
0, 3: ●
1, 4: ●
…
6, 3: ●
6, 4: ●
```

GraphBLAS MatrixRange Example

Iteration over the stored elements (the long form).

```
template <grb::MatrixRange M>
void output_entries(M&& A) {
  for (auto iter = A.begin();
       iter != A.end();
       ++iter) {
    auto&& [location, value] = *iter;
    auto&& [i, j] = location;
    cout << i << ", " << j << ": " << value << endl;
  }
}
```



Possible output:

```
0, 1: ●
0, 3: ●
1, 4: ●
…
6, 3: ●
6, 4: ●
```

Generic Algorithms using the MatrixRange Concept

Matrix reduction:

```
template <grb::MatrixRange M>
grb::matrix_scalar_t<M> sumreduce(M&& A)
{
   grb::matrix_scalar_t<M> sum = 0;
   for (auto&& [location, v] : A) {
      sum += v;
   }
}
```

Sparse times dense matrix multiply:

```
template <grb::MatrixRange M,
          class          T>
void spmm(M&& A, size_t N,
          std::vector<T> const &B,   // dense
          std::vector<t>       &C) { // dense, cleared
   for (auto&& [location, a_ik] : A) {
      auto&& [i, k] : location;
      for (size_t j = 0; j < N; ++j) {
         c[i*N + j] += a_ik * B[k*N + j];
   }
}
```

Views

- They provide a **lazily evaluated view** (or interface) to some data
- We can **apply transformations (lazily)** without copying
- C++ ranges library defines a **collection of views**, such as transform, filter, etc.
- GraphBLAS defines a collection of views, such as **transpose** and **complement**

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

// Create lazily evaluated view of Aᵀ
auto A_t = grb::views::transpose(A);

auto C = grb::multiply(A, A_t);
```

Views

- They provide a **lazily evaluated view** (or interface) to some data
- We can **apply transformations (lazily)** without copying
- C++ ranges library defines a **collection of views**, such as transform, filter, etc.
- GraphBLAS defines a collection of views, such as **transpose** and **complement**

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

// Create lazily evaluated view of Aᵀ
auto A_t = grb::views::transpose(A);

auto C = grb::multiply(A, A_t);
```

Views are also defined to transform **external data** to conform to GraphBLAS concepts like **MatrixRange.**

Adapting External Graph Data Structures

- NWGraph edge lists
- NWGraph adjacency lists
- CSR C-arrays

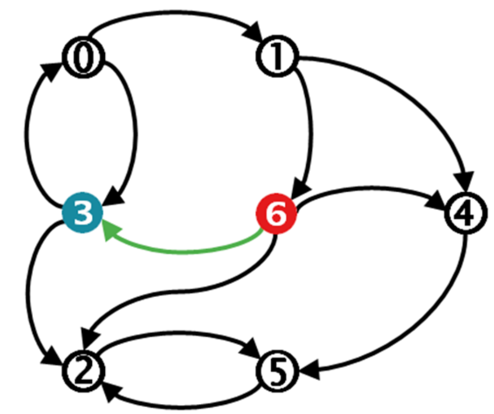
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

NWGraph

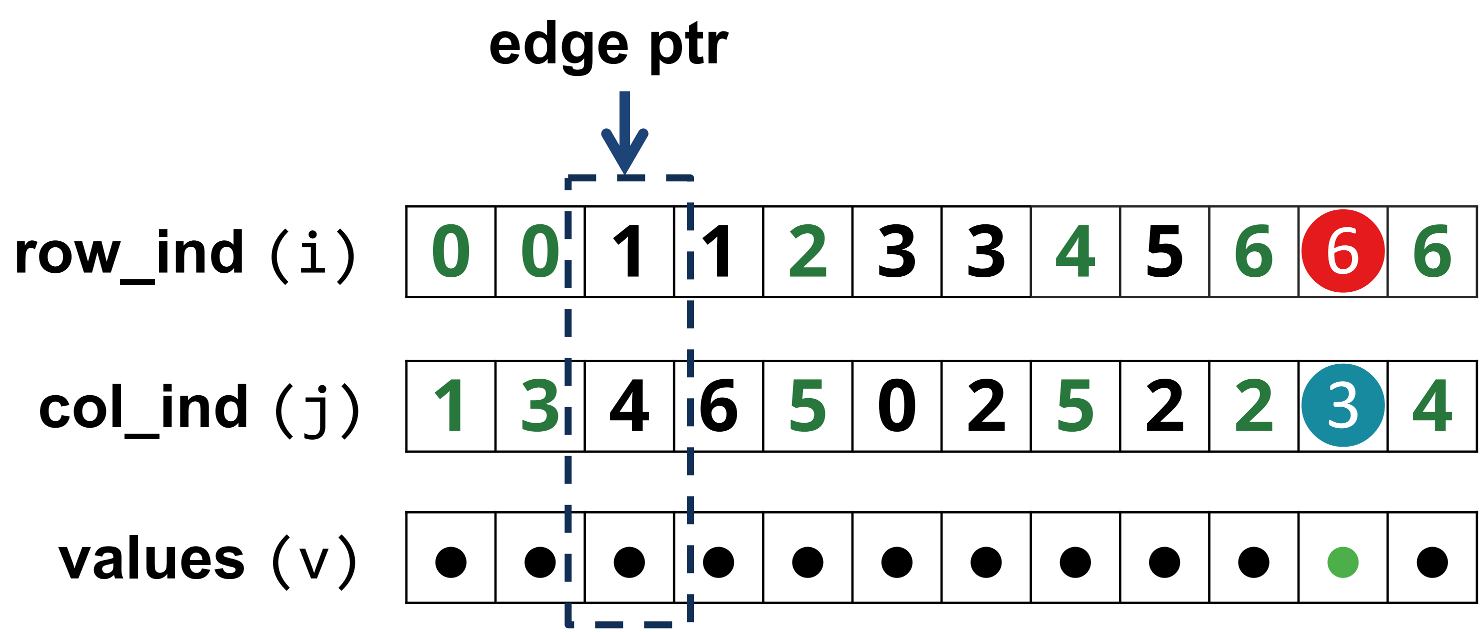
- A library of **generic algorithms** and **data structures** for graph computation
- Uses C++20 and modern C++ techniques
- Supports shared memory parallelism
- Strongly influencing the C++ Graph Library Standard proposal (P1709)
- NWGraph concepts describe different patterns for **iterating through a graph**
  - `edge_list_graph` (e.g., COO data structures)
  - `adjacency_list_graph` (e.g., CSR/CSC data structures)

Repository: https://github.com/pnnl/NWGraph
Paper: Lumsdaine, Andrew, et al. "NWGraph: A Library of Generic Graph Algorithms and Data Structures in C++ 20." In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

Edge Lists (like COO format)




```
template <class index_type,
          class scalar_type>
auto sumreduce(size_t       num_edges,
               index_type  *row_ind,
               index_type  *col_ind,
               scalar_type *values)
{
   scalar_type sum = 0;
   for (auto e = 0; e < num_edges; ++e) {
      /*index_type i = row_ind[e]; */
      /*index_type j = col_ind[e]; */
      sum += values[e];
   }
   return sum;
}
```



NWGraph Edge List

- NWGraph's **`edge_list_graph`** requires **one-dimensional iteration** through "container" of edges (3-tuples).
  - Minimum requirement: forward iteration
- The value type of a data element is a triplet:

  `{src, dst, value}`

- Remember: Any data structure that supplies the correct interface **satisfies the concept.**

```
template <nw::graph::edge_list_graph G>
auto sumreduce(G&& g) {
   float sum = 0;

   for (auto&& [i, j, v] : g) {
      sum += v;
   }

   return sum;
}
```

Adapting NWGraph’s Edge List Graph

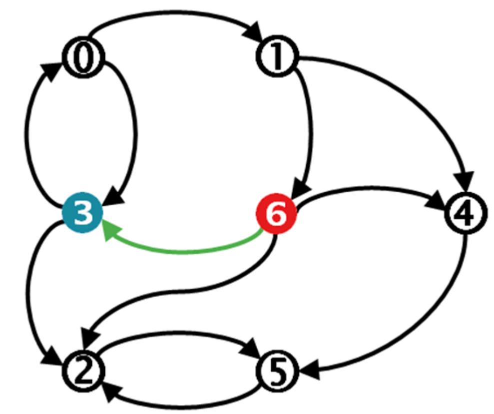
- Edge lists are already **flattened** (1-dimensional iteration) data structures.
- Adapting with a GraphBLAS view only requires **restructuring** of the data elements:

```
{i, j, v} → {{i, j}, v}
```

- The **forward range** portion of the view is implemented using pipe (“|”) syntax from **C++ ranges library**’s *range adaptors:*

```
template <nw::graph::edge_list_graph G>
auto transform_range(G&& graph) {
  return graph
          | std::views::transform(
              [](auto&& edge_entry) {
                auto&& [i, j, v] = edge_entry;
                return grb::matrix_ref(grb::index(i,j), v);
              });
}
```

Adjacency Lists (CSR-like format)



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

0	1,●	3,●	
1	4,●	6,●	
2	5,●		
3	0,●	2,●	
4	5,●		
5	2,●		
6	2,●	3,●	4,●

row ptr (outer)

out edge ptr (inner)

NWGraph Adjacency List

- The **`adjacency_list_graph`** concept defines support for **hierarchical** iteration:

  "*…a random-access range of forward ranges*."

- The "outer" iterator steps through **vertices** (row of adjacency matrix).
  - Value type is a "forward range of out edges"
  - Vertex id (i) is implicit

- The "inner" iterator steps through **out edges** of the corresponding vertex (elements of the row)

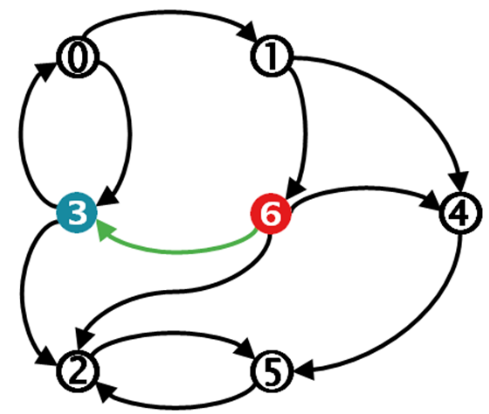
```
template <nw::graph::adjacency_list_graph G>
auto sumreduce(G&& g) {
   float sum = 0;
   // index   i = 0;
   for (auto&& out_edges : g) {
      for (auto&& [j, v] : out_edges) {
         sum += v;
      }
      // ++i;
   }
   return sum;
}
```

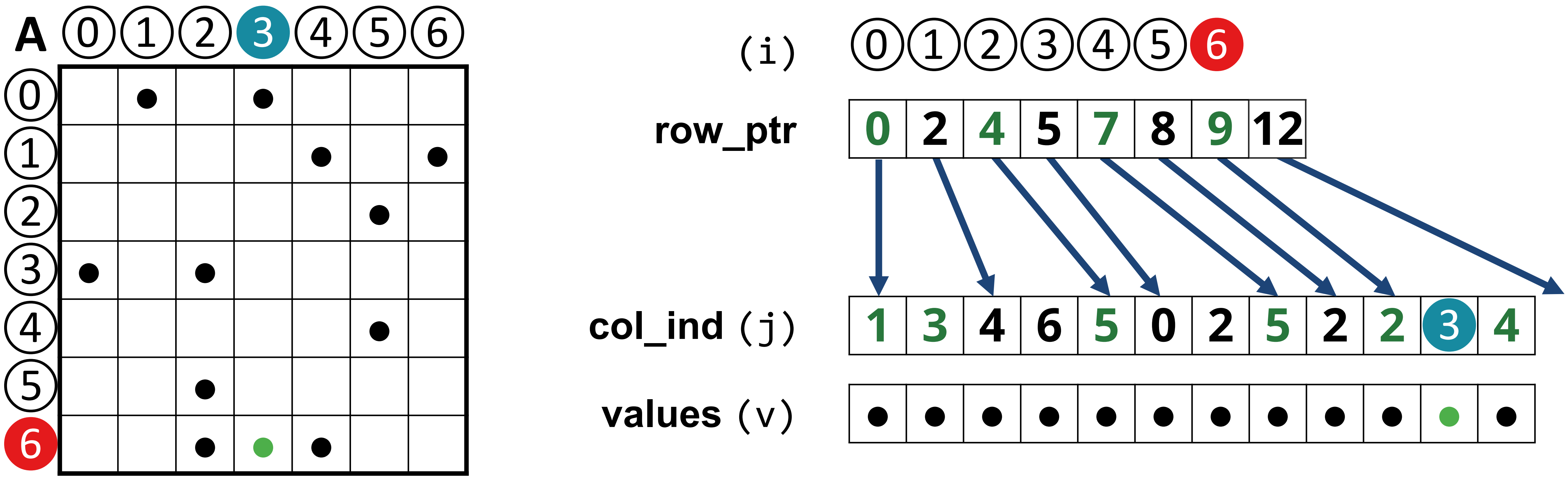
Example: Adapting NWGraph's Adjacency List Graph

- The **forward range** portion of the view adaptor is shown to the right
- Adapting them requires:
  1. Adding the implicit row id
  2. Traversing both ranges
  3. Restructuring of the data elements.
  4. Flattening of the nested iteration.

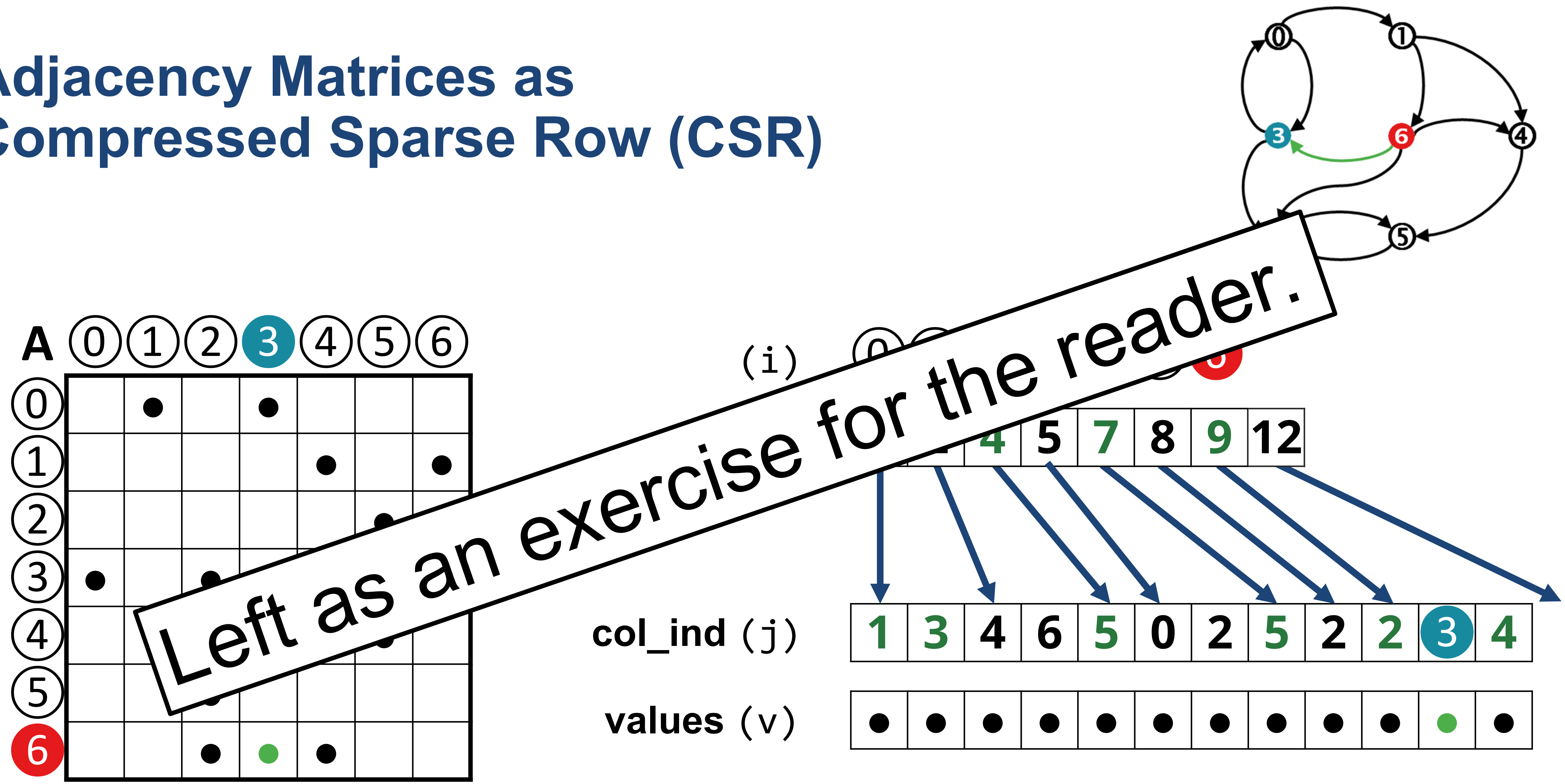
```
template <nw::graph::adjacency_list_graph G>
auto transform_range(G&& graph) {
  return graph
         | enumerate()         // (0, row[0]), (1, row[1]), ...
         | std::views::transform(
             [](auto&& row_entry) {
               auto&& [i, row] = row_entry;
               return row
                      | std::views::transform(
                        [i](auto&& entry) {
                          auto&& [j, v] = entry;
                          return grb::matrix_ref(
                                   grb:index_type(i, j), v);
                        });
             })
         | std::views::join;  // flattens here (joins all rows)
}
```

Adjacency Matrices as Compressed Sparse Row (CSR)





Adjacency Matrices as Compressed Sparse Row (CSR)



Evaluation

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Experimental Setup

- Two common GraphBLAS operations:
  - Matrix reduction (to scalar)
  - SPMM: sparse times dense matrix
  - Assumptions:
    - Numeric data type (float)
    - Arithmetic Semiring
- Platform
  - Dual Intel® Xeon® Platinum 8480+, 2GHz
  - 512GB RAM
  - GCC 12.2.0, -O3, -march=sapphirerapids
  - **Single thread**
- GOAL: Measure the overhead of using views relative to “native” code

```
template <grb::MatrixRange M>
grb::matrix_scalar_t<M> sumreduce(M&& A)
{
   grb::matrix_scalar_t<M> sum = 0;
   for (auto&& [location, v] : A) {
      sum += v;
   }
}
```

```
template <grb::MatrixRange M,
          class          T>
void spmm(M&& A, size_t N,
          std::vector<T> const &B,   // dense
          std::vector<T>       &C) { // dense/cleared
   for (auto&& [location, a_ik] : A) {
      auto&& [i, k] : location;
      for (size_t j = 0; j < N; ++j) {
         c[i*N + j] += a_ik * B[k*N + j];
   }
}
```

Experimental Setup: Input matrices

- Sparse matrices used in the evaluation:
  - Shape: m x m
  - All very sparse (98.5% - 99.999% sparse).
- Dense matrices:
  - Shape: m x 32
  - Contiguous array of elements

Sparse Matrix	Kind	$m = k$	NNZ	CSR Size	COO Size
com-Orkut	NMF	3.1M	234M	2.8 GB	5.6 GB
ldoor	Structural	952K	46.5M	565 MB	1.1 GB
Mouse Gene	Biology	45.1K	29M	348 MB	695 MB
nlpkkt160	NLP	8.3M	230M	2.8 GB	5.5 GB
kim2	2D Mesh	457K	11.3M	140 MB	272 MB

Experimental Setup: Data structures

- **“GraphBLAS Native CSR”**
  - Reference Library’s (RGRI) implementation of `grb::matrix`
  - Three contiguous arrays
  - Using the generic MatrixRange interface elements only (i.e., **not tuned** for CSR)
- **“CSR (View)”**
  - Three C-style arrays
  - Adapted to MatrixRange with a view
- **NWGraph’s `edge_list`** and **`adjacency_list`** data structures
  - “**(Direct)**” – native performance using NWGraph library directly
  - “**(View)**” – NWGraph data structure through a MatrixRange view adaptor

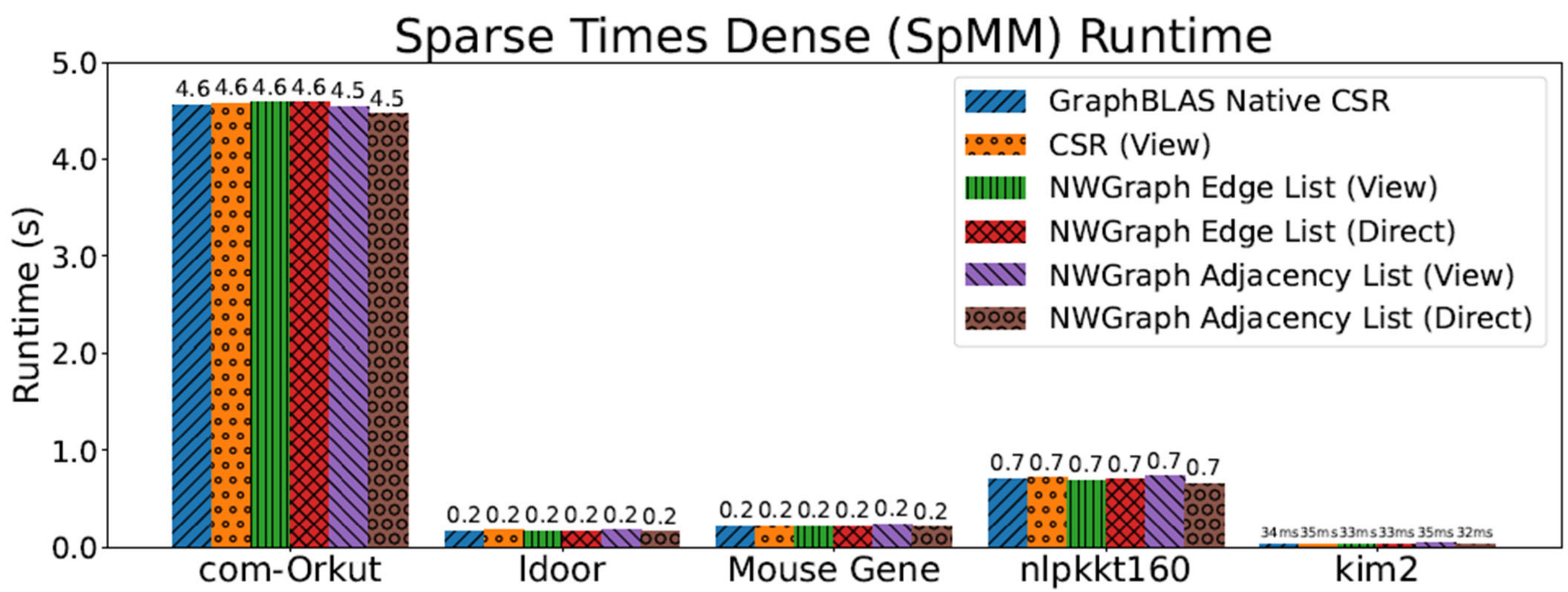
```
// Run “CSR (View)” experiment
uint32_t  m = ..., n = ...;
size_t    nnz = ...;
size_t   *row_ptr = ...;
uint32_t *col_ind = ...;
float    *values  = ...;

auto a_view = grb::csr_matrix_view(values,
                                   row_ptr,
                                   col_ind,
                                   {m, n}, nnz);

grb::spmm(a_view, b, c);
auto d = grb::sumreduce(a_view);
```

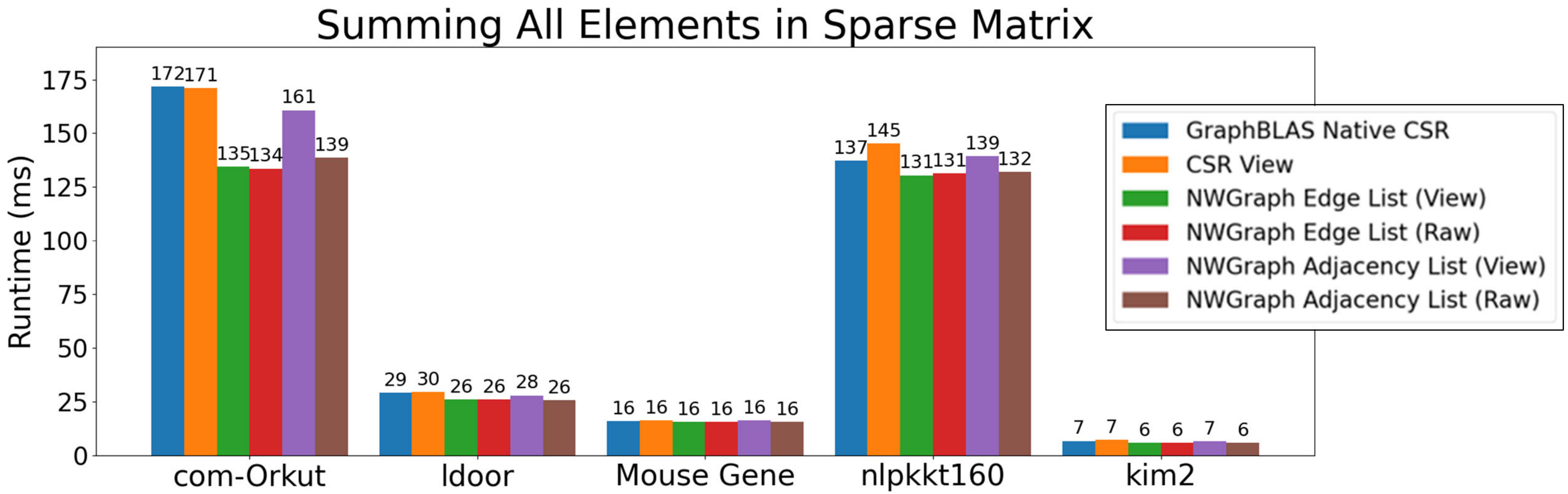
Experimental Results: SPMM

- Little to no overhead in adapting other data structures to MatrixRange
- The amount of computational work hides the overheads



Experimental Results: Matrix reduction (sumreduce)

- Less computational intensity shows overhead of flattening hierarchical data structures
- Up to 15% overhead (when adapting NWGraph Adjacency List)



Conclusions

- Defined **concepts** and implemented **views** (adaptors) for many different data structures
- The approach works with acceptable amounts of overhead (5 – 15%)
  - Detailed analysis of the assembly code generated gives insight to possible improvements.
- Caveat: results are specific to these datasets and workloads.
  - Some applications may benefit from copying the data in or out
  - C++ API specification will still include **import** and **export** of data

Future Work

- Avoiding **explicit constructor calls** for views
    - Adding another CPO would allow for automatic discovery of supported views
    - C++ ranges library has automatic view support through a `grb::views::all`
    - Find some of this work in the RGRI repository
- **Multi-dimensional iteration** (discussed in last year's GrAPL paper)
    - Row views and nested iterators (like in NWGraph)
    - Avoids flattening (hampers compiler optimization)
    - Deferred to a later release of the C++ API Specification
        - What should be supported?
        - What will be offered in future releases of the C++ Standard Library
- Concepts for ordered iteration
- Views for **mutating data** (i.e., for MutableMatrixRange)

Request for Comments

- First draft of the GraphBLAS C++ API Specification nearing completion
  - Depends on (but does not include) the mathematical specification of each operation
  - Plans underway to extract a math specification from the C API for both APIs

- Interested parties may review and comment on the C++ Specification
  - Repository: https://github.com/GraphBLAS/graphblas-api-cpp
  - **Use github Issues provide feedback and request changes/additions**

- Reference Implementation is underway (where these experiments were performed):
  - Repository: https://github.com/GraphBLAS/rgri