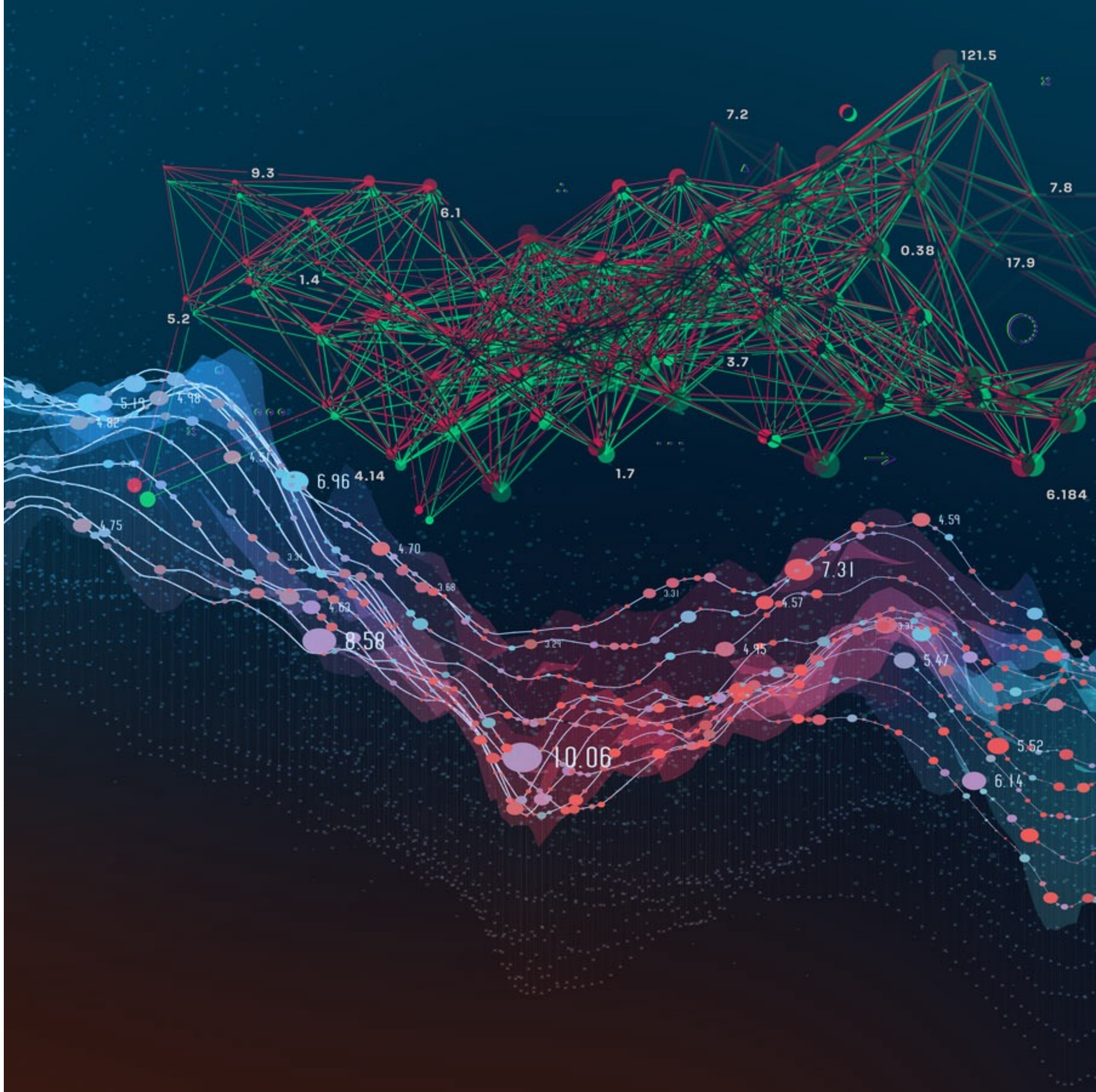
Automatic Code Generation for High-Performance Graph Algorithms

The Seventh LLVM Performance Workshop,
CGO, Montreal, Canada, February 25, 2023

Rizwan Ashraf, Zhen Peng,
Luanzheng Guo, Gokcen Kestor

Pacific Northwest National Laboratory

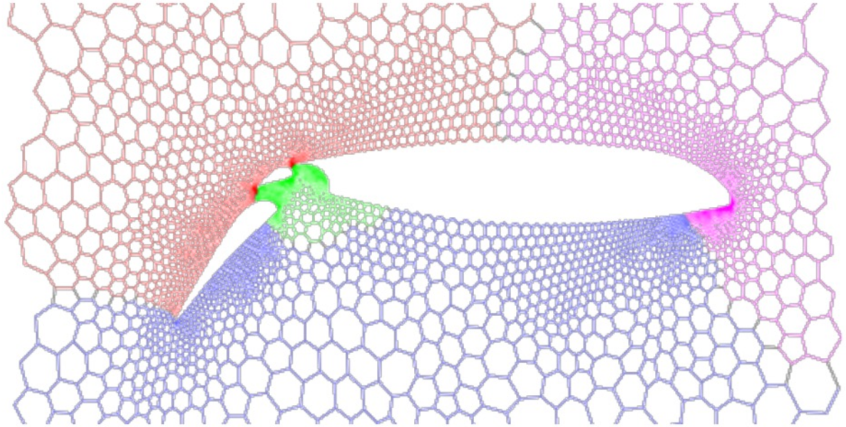
PNNL is operated by Battelle for the U.S. Department of Energy



Graph Algorithms

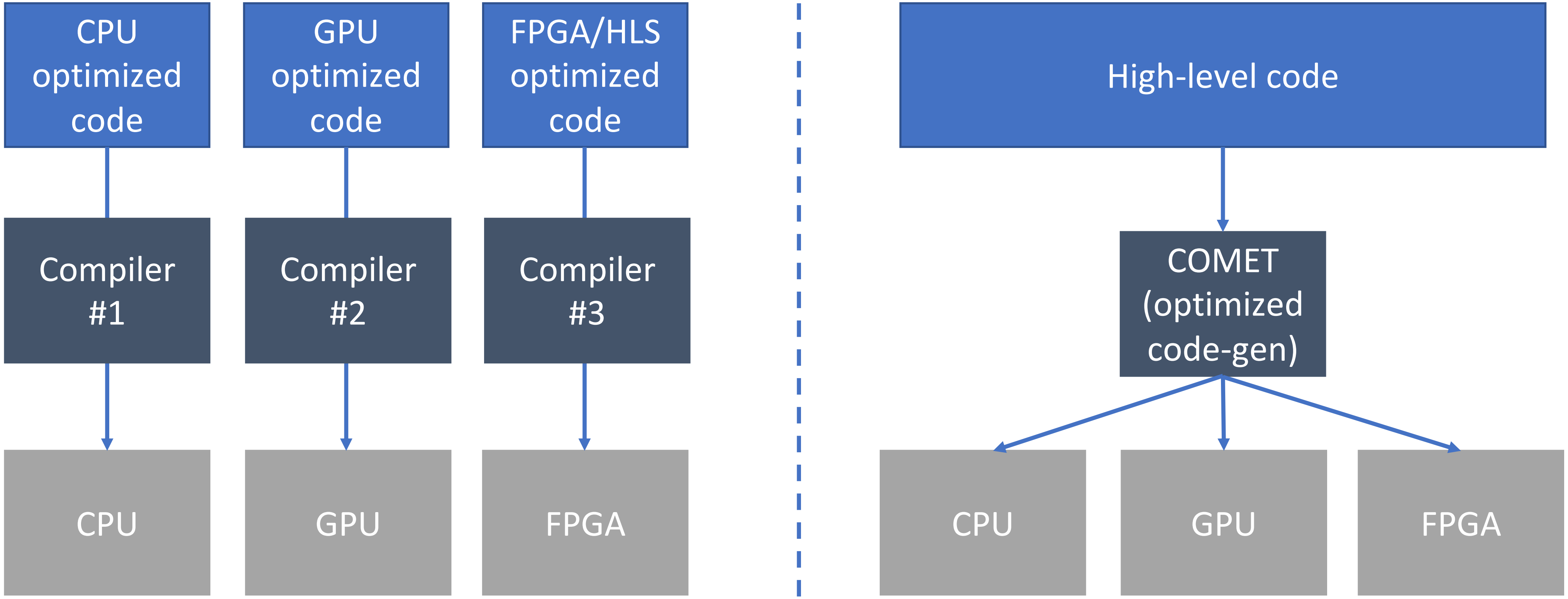
- The use of graph processing is everywhere around us!
  - Social Networks: recommendation systems
  - Travel: Shortest paths, food/hotel recommendations, etc.
  - …
  - Scientific Computing: Biology (genome assembly, human brain), Power Grid, Load Balancing.

- There are a variety of graph algorithms
  - Graph libraries exist for various targets.
  - We propose a compiler approach.



Picture Credit: Sanders and Schulz

The Compiler Approach



CPU optimized code → Compiler #1 → CPU

GPU optimized code → Compiler #2 → GPU

FPGA/HLS optimized code → Compiler #3 → FPGA

3X effort for writing libs (in most cases)

High-level code → COMET (optimized code-gen) → CPU, GPU, FPGA

Write once, run anywhere

COMET: COMpiler for Extreme Targets

- Porting graph applications to heterogeneous systems often requires porting code to different programming environments.
  - Explosion of complexity and versioning.
  - Difficult to achieve performance portability.
- For performance portability, need to identify computational patterns.
  - High-level languages allow users to express high-level computational patterns/motifs.
  - Semantics information is used for efficient code generation.
- Clear separation of responsibilities.
  - Users implement algorithms using high-productive programming environments.
  - Compiler generates efficient code for heterogeneous architectures.

The Challenge with Graph Algorithms

- Traditional architectures are mostly designed for structured data accesses (lists, stacks, etc.).
  - Graph algorithms operate on irregular sparse data.
  - The time spent in communication is high as compared to computation.
  - Conventional latency hiding techniques do not provide much benefit.
- Challenge to program efficient graph algorithms
  - Random access patterns provides poor locality in cache, and hence lot of misses.
  - Parallelization is difficult.
- Optimizations if performed severely limit portability of graph algorithms to new architectures.
  - Compilers can help, but we also need a new way of doing graph algorithms.

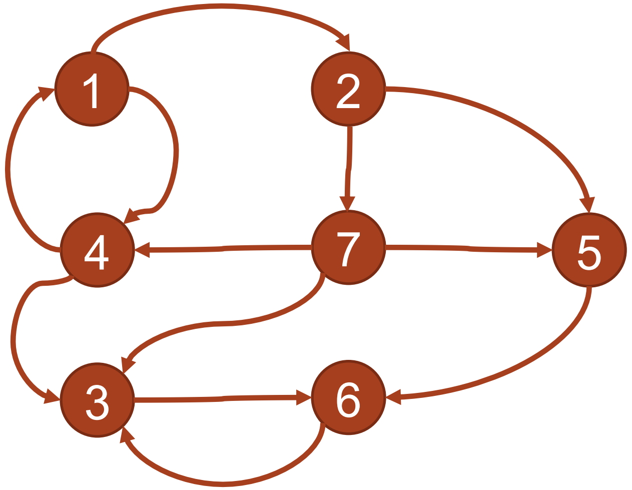
Graph Algorithms in Linear Algebra

- Linear algebra (LA) provide an elegant, concise, intuitive, and portable programming abstraction for implementing graph algorithms.
    - The algorithmic complexity of LA-based implementations is close to the complexity of the node- or edge-traversal-based implementations.
- Linear algebra operators have been extensively studied and optimized for a variety of architectures and domain problems
    - Many algorithmic implementations of operators and methods
    - Many LA accelerators exists (e.g., Tensor cores)
    - Good support in many architectures (e.g., AVX)
    - LA operators represent basic computational blocks in emerging architecture

LA-based Graph Processing

- A graph as a sparse adjacency matrix.
- Sparse matrix/vector operations can be used to express graph algorithms.

	1		1			
				1		1
					1	
1		1				
					1	
		1				
		1	1	1		



LA-based Graph Processing

- A graph as a sparse adjacency matrix.
- Sparse matrix/vector operations can be used to express graph algorithms.
    - Find vertices that are one hop away from a source vertex: fA

Only operations are performed where non-zero elements exist

A

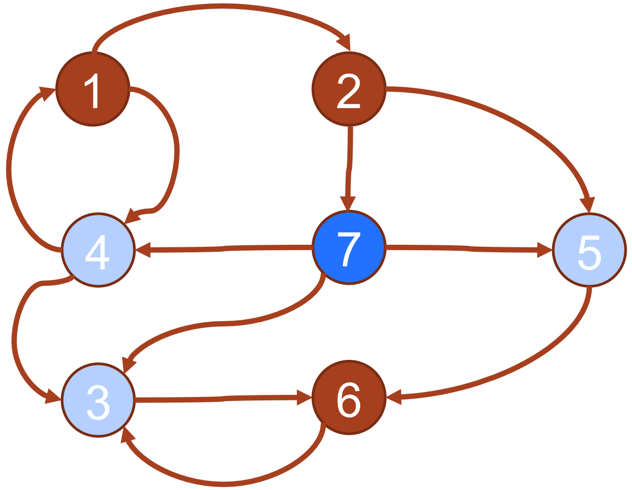
	1		1			
				1		1
					1	
1		1				
					1	
		1				
		1	1	1		

Frontier, f

						1

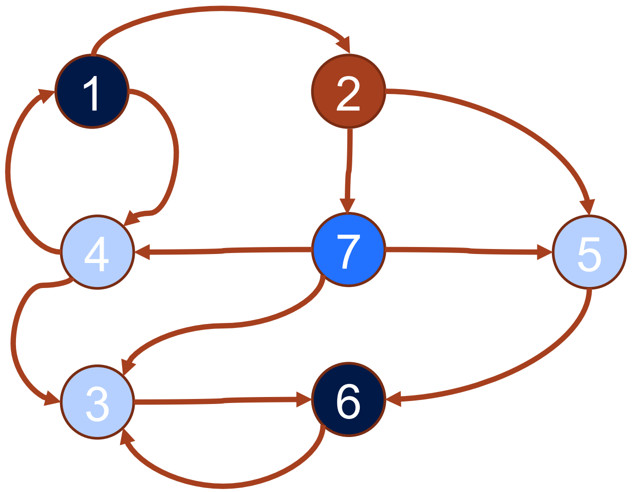
		1	1	1		

fA



LA-based Graph Processing

- A graph as a sparse adjacency matrix.
- Sparse matrix/vector operations can be used to express graph algorithms.
  - Find vertices that are k hops away from a source vertex: fA^k



A

	1		1			
				1		1
					1	
1		1				
					1	
		1				
		1	1	1		

	1		1			
				1		1
					1	
1		1				
					1	
		1				
		1	1	1		

						1

Frontier, f

		1	1	1		

fA

1		1			2	

fA^2

Semirings

- A semiring is an algebraic structure that allows us to perform special operations beyond addition and multiplication to elements in a generic matrix multiplication operation.

Any-pair semiring for traversal. Operates on binary values (structure).

Frontier f

			1		1	

A

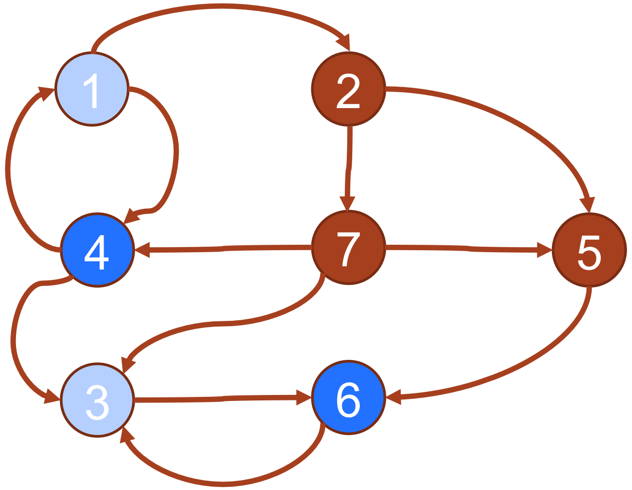
	1		1			
				1		1
					1	
1		1				
					1	
		1				
		1	1	1		

Any (pair(1,1), pair(1,1))

1		1				

$w = f@(any, pair)A$

$f@(+,\times)\ A$



Semirings

- A semiring is an algebraic structure that allows us to perform special operations beyond addition and multiplication to elements in a generic matrix multiplication operation.

Plus-times semiring for traversal. Operates on natural numbers.

A

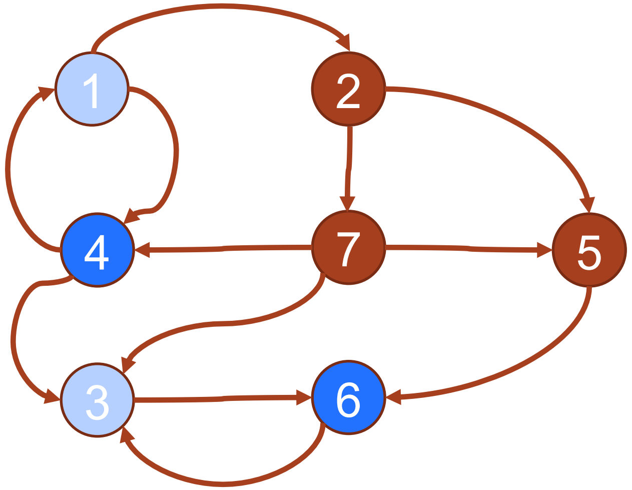
	1		1			
				1		1
					1	
1		1				
					1	
		1				
		1	1	1		

Frontier f

			1		1	

1		2				

$w = f@(+,\times)\,A$



Semirings

- A semiring is an algebraic structure that allows us to perform special operations beyond addition and multiplication to elements in a generic matrix multiplication operation.

Min-Plus semiring for finding shortest path. Operates on +Real numbers.

Frontier f

			0.5		0.6	

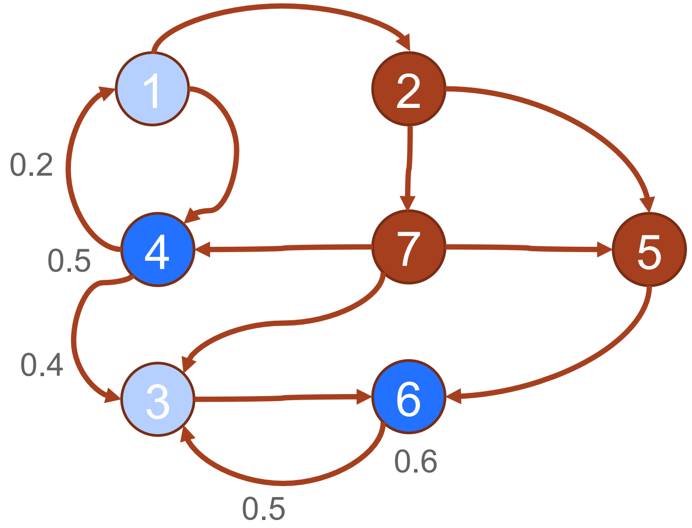
A

	1		1			
				1		1
					1	
0.2		0.4				
					1	
		0.5				
		1	1	1		

Min (0.5+0.4, 0.6+0.5)

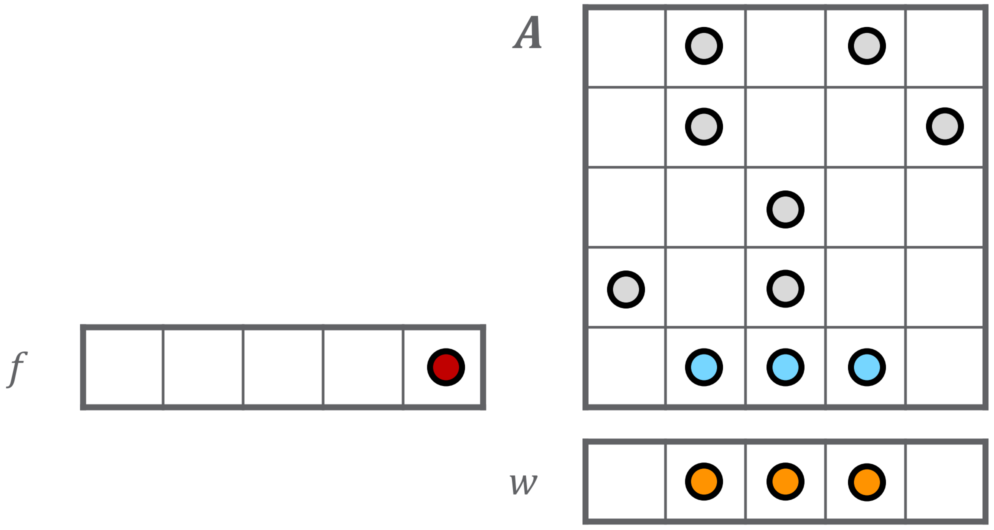
0.7		0.9				

$w = f@(min, +)A$



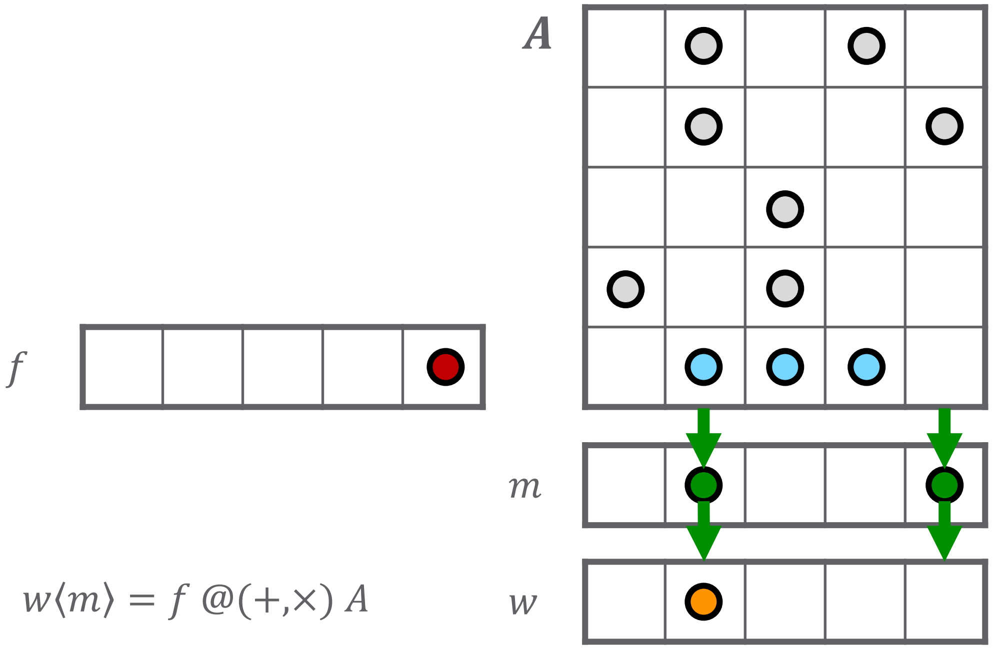
Masking

- Prevent redundant computations (traversal: already visited vertices)
- Reduce the scope of an operation to be performed
  - A mask indicates the locations where the operation should be performed



$w = f\ @(+,\times)\ A$

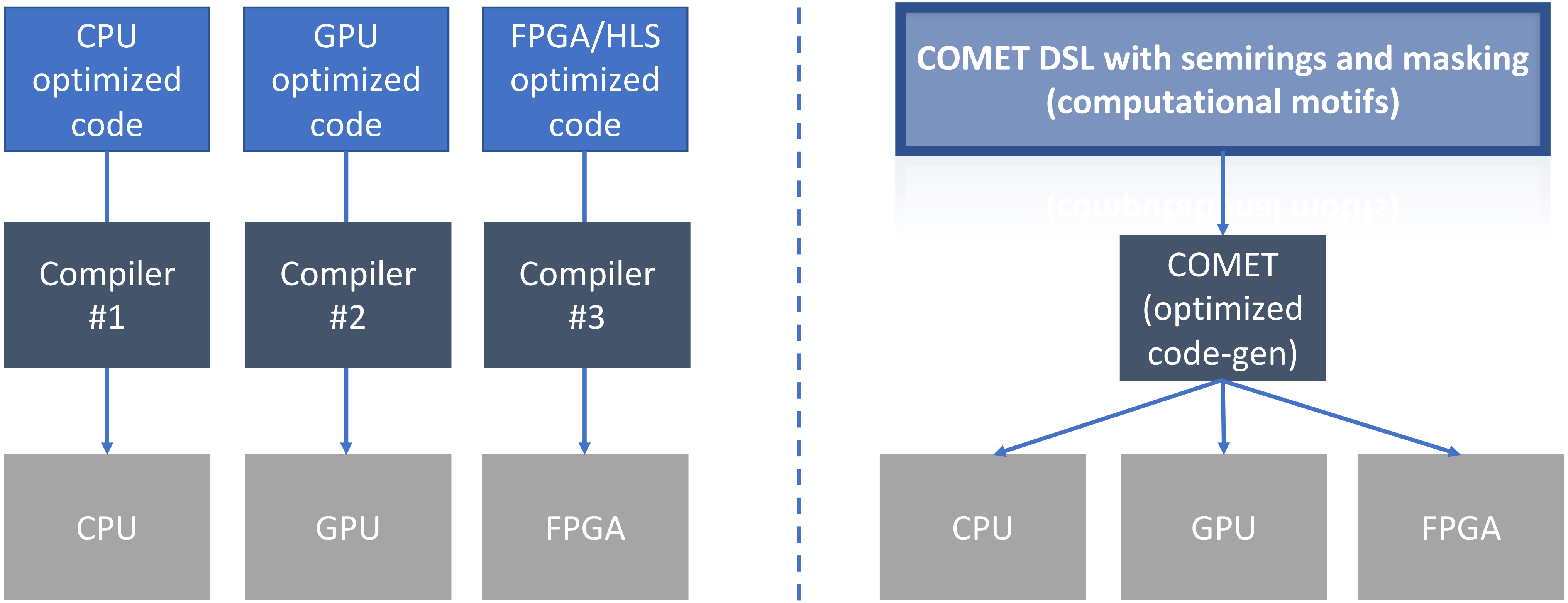
without a mask



$w\langle m\rangle = f\ @(+,\times)\ A$

with a mask

A DSL for Graph Algorithms using Linear Algebra



CPU optimized code

GPU optimized code

FPGA/HLS optimized code

Compiler #1

Compiler #2

Compiler #3

CPU

GPU

FPGA

3X effort for writing libs (in most cases)

COMET DSL with semirings and masking (computational motifs)

COMET (optimized code-gen)

CPU

GPU

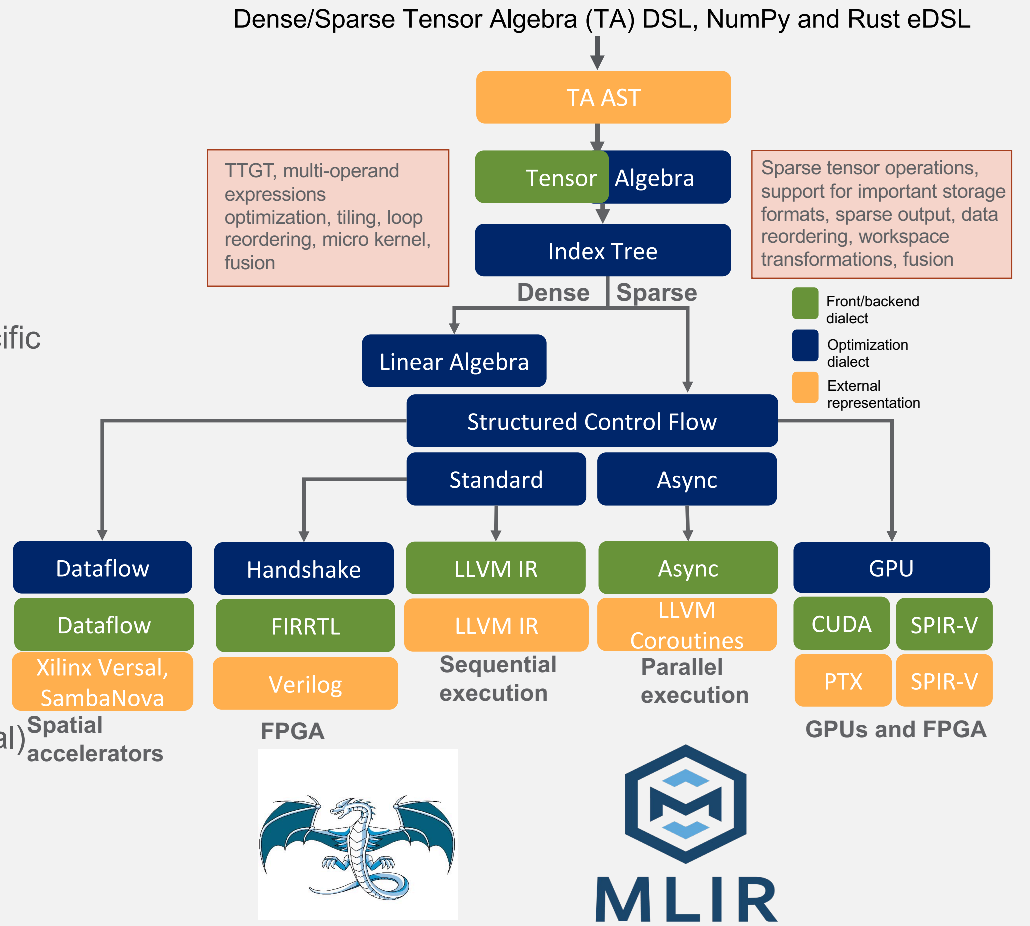
FPGA

Write once, run anywhere

COMET: Domain Specific Compilation in Multi-level IR

- COMET is a compiler infrastructure that focuses on computational chemistry and graph analytics application domain
- COMET supported frontends
  - COMET Domain specific language that follows Einstein notation
  - NumPy einsum to evaluates the Einstein summation
  - Rust eDSL
- COMET compiler infrastructure
  - Enable from high-level, domain-specific and low-level, architecture-specific compiler optimizations
  - **Tensor algebra dialect** in the MLIR infrastructure
  - **Multi-level** code optimizations, including domain-specific and architecture specific
  - **Abstraction** for dense/sparse storage formats
    - ✓ A set of per-dimension attributes to specify sparsity properties of tensors
    - ✓ Attributes enables support for **a wide range of sparse storage formats**
  - Data layout optimizations to enhance **data locality**
  - Support for **sparse output** for sparse-sparse computation (e.g., SpGEMM)
  - Support for **semiring** operations to represent graph algorithms
  - **Kernel Fusion** to avoid temporaries and redundant computation
  - Automatic code generation for **sequential** and **parallel** execution
  - **FPGA** code generation via SPIRV binary
  - Interface with **emerging dataflow architectures** (SambaNova and Xilinx Versal)
- COMET runtime
  - **Input-dependent optimization** to increase data locality and load balancing
  - Read input matrices and tensors, convert it into internal storage format

$$C_{ijkl} = \sum_{mn} A_{imkn} \cdot B_{jnlm}$$



Multi-Level Intermediate Representation (MLIR)

A collection of modular and reusable software components that enables the progressive lowering of high-level operations, to efficiently target hardware in a common way

New compiler infrastructure

Part of LLVM project

https://github.com/llvm/llvm-project

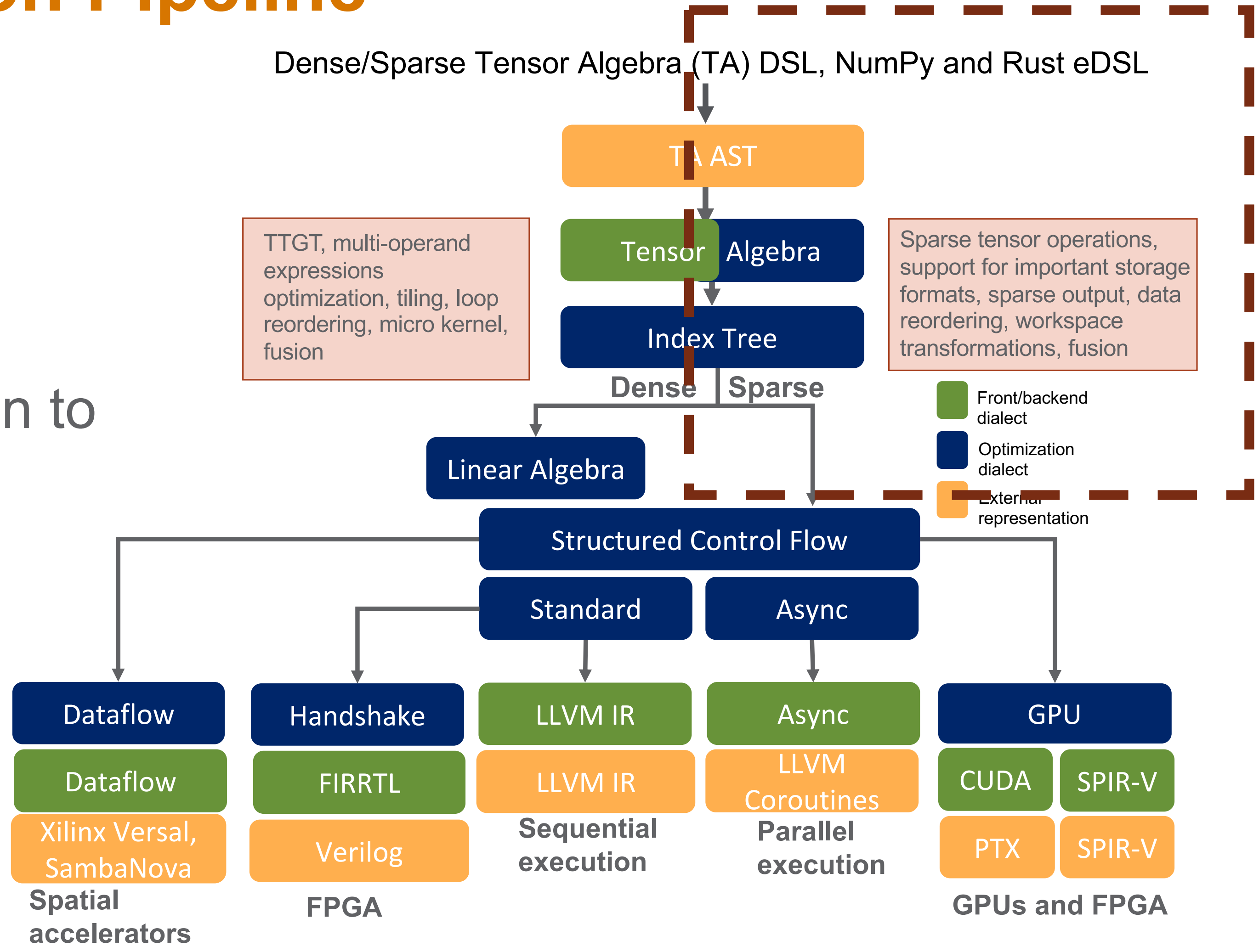
Sparse Computations

- Sparse kernels are widely used in many applications, e.g., scientific computing, machine learning, and data analytics
- Sparse computations uses sparse storage formats:
  - To reduce storage requirements by storing only nonzero elements
  - To reduce computational requirements by skipping redundant computation
- Challe
  - Dif
  - Lack of temporal locality due to irregular accesses
  - Lack of spatial locality, limited data reuse
- Sparse libraries solve some of the issues above but …
  - Limited support for combination of sparse storage formats, various tensor expressions, and heterogeneous target architectures

Sparse Compilers simplifies development of sparse kernels by automatically generating code based on tensor “***sparsity”*** property

Sparse Compilation Pipeline[1,2]

- Internal sparse tensor storage format
- Sparse data type
- An attribute per tensor dimension to support sparse tensor storage format
- Automatic code generation for sparse tensor operations
- Support for sparse output
- Input-dependent optimization
  - Data reordering to enhance data locality



Dense/Sparse Tensor Algebra (TA) DSL, NumPy and Rust eDSL

TA AST

Tensor Algebra

Index Tree

Dense | Sparse

TTGT, multi-operand expressions optimization, tiling, loop reordering, micro kernel, fusion

Sparse tensor operations, support for important storage formats, sparse output, data reordering, workspace transformations, fusion

🟩 Front/backend dialect

🟦 Optimization dialect

🟧 External representation

Linear Algebra

Structured Control Flow

Standard

Async

Dataflow → Dataflow → Xilinx Versal, SambaNova — **Spatial accelerators**

Handshake → FIRRTL → Verilog — **FPGA**

LLVM IR → LLVM IR — **Sequential execution**

Async → LLVM Coroutines — **Parallel execution**

GPU → CUDA, SPIR-V → PTX, SPIR-V — **GPUs and FPGA**

---

[1] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, Gokcen Kestor. “A High Performance Sparse Tensor Algebra Compiler in MLIR”. *LLVM-HPC*, 2021.
[2] Sparse tensor algebra optimizations in MLIR. Tian R., L. Guo, and G. Kestor. 2021 LLVM DEVELOPERS' MEETING. November 2021.

Support for Sparse Outputs

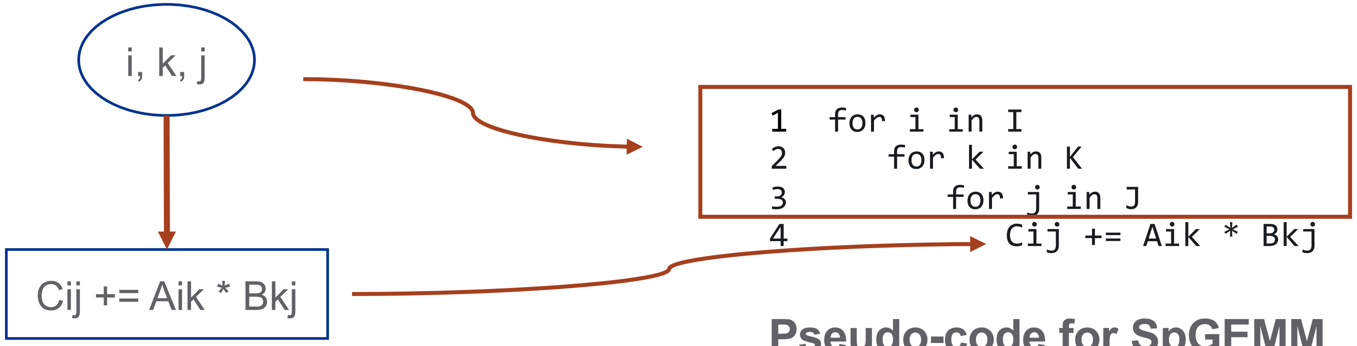
- Storing output tensor in a sparse format introduces expensive insertions and accesses to sparse input tensors, which has large time complexity
- We introduced a **temporary dense data structure** (called workspaces[1]) to store the value in the sparse dimension in sparse kernels to improve **data locality** of sparse kernels while producing **sparse output**
- This approach brings the following advantages:
  - Significantly improves performance of sparse kernels through efficient dense data structures accesses.
  - Reduces memory footprint
  - Avoids “densifying” issue in the compound expressions

[1] Tensor Algebra Compilation with Workspaces. Fredrik Kjolstad, and et al., IEEE/ACM International Symposium on Code Generation and Optimization, 2019

Index tree Intermediate Representation (IR)

- We introduced ***Index Tree*** intermediate representation in the COMET compiler
  - Index Tree is a high-level intermediate representation for a tensor expression
  - Consists of two types of nodes
    - ✓ **Index nodes:**
      - Contain one or more indices to represent (nested) loops
      - Each index represent a level of loop
    - ✓ **Compute nodes:**
      - Contain compute statements

$$Cij = \sum_{k} Aik \ * Bkj$$



```
for i in I
   for k in K
      for j in J
         Cij += Aik * Bkj
```

Index tree for SpGEMM

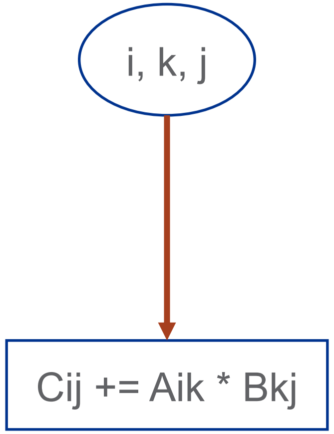
Pseudo-code for SpGEMM

Workspace Transformation

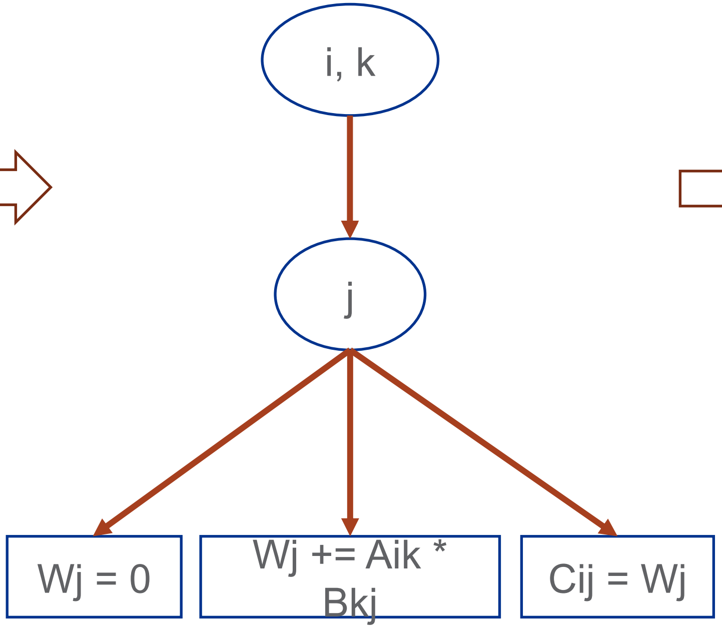
- We perform compiler transformation in the index tree representation of a tensor expression
  - Benefits
    - ✓ Reduces expensive insertions/ accesses to sparse tensors
      - Dense data structure has better locality
      - Generates “for” loops instead of “while” loops
      - Utilize the existing for loop optimizations
  - How?
    - ✓ Identify the index that needs workspace
      - Store the value in the dimension into workspace (i.e., dense low dimensional data structure)
    - ✓ Check output tensor (lhs) , if it contains sparse dimension
      - e.g., SpGEMM in CSR, dimension j is sparse in C. Then the original “Cij=Aik*Bkj” will be transformed into “Wj = 0; Wj += Aik*Bkj; Cij = Wj; ” in each iteration of i
    - ✓ Check input tensors (rhs), if one dimension in both two input tensors are sparse
      - e.g., pure sparse elementwise multiplication, Cij=Aij*Bij, all matrices are in CSR. In this case, dimension j is sparse in A and B. then the original “Cij=Aij*Bij” will be converted into “Wj = 0;Wj = Aij; Cij = Wj*Bij;” in each iteration of i

What is the algorithm?

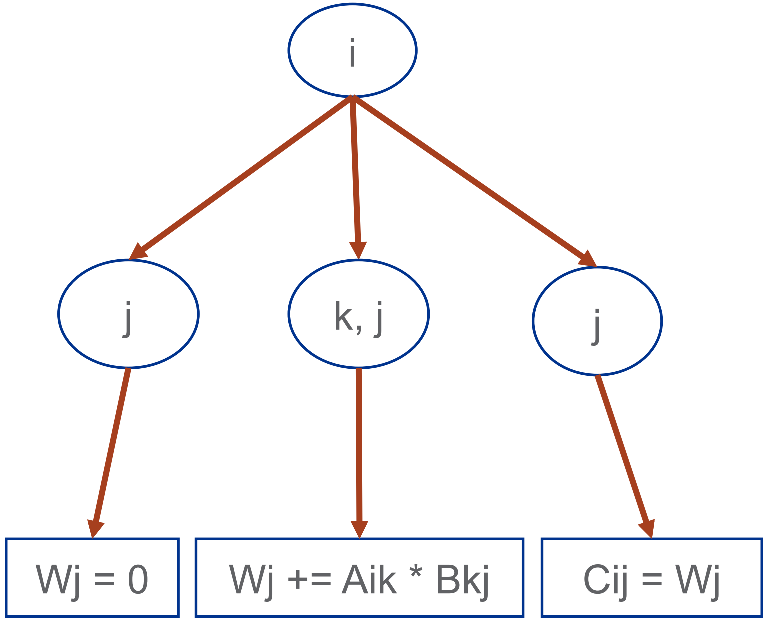
Transformation in Index Tree



Index tree for SpGEMM

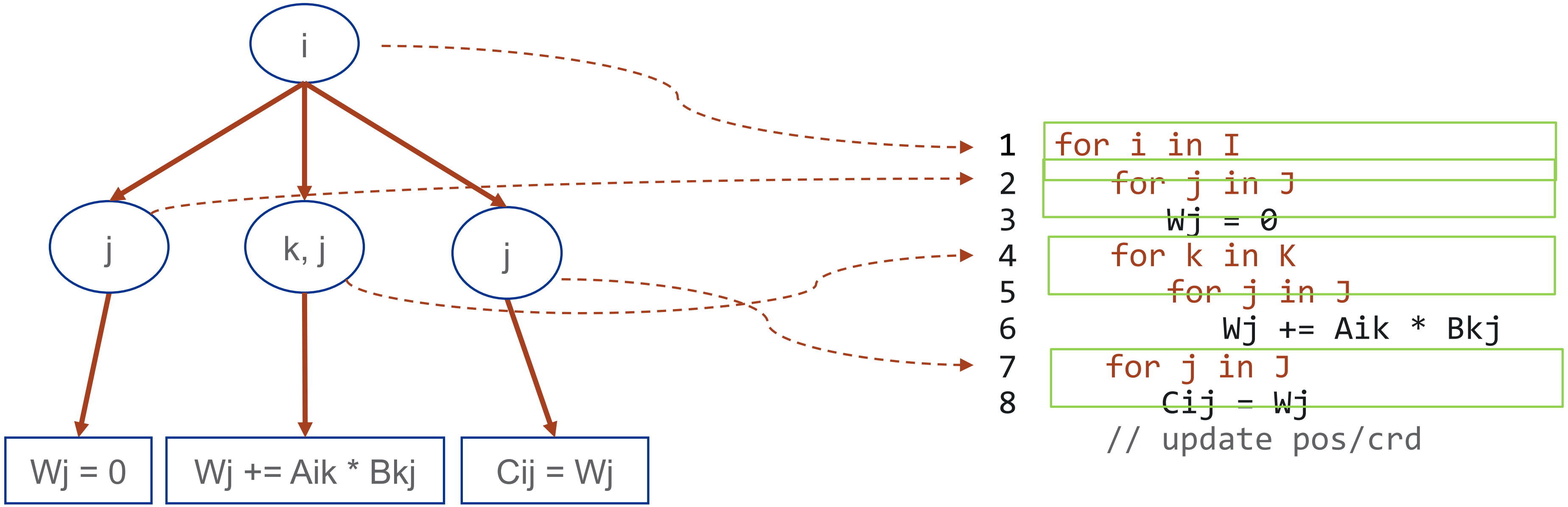


Index tree for SpGEMM with workspace



Eliminate loop invariant redundancy

Code Generation from Index Tree IR Operations



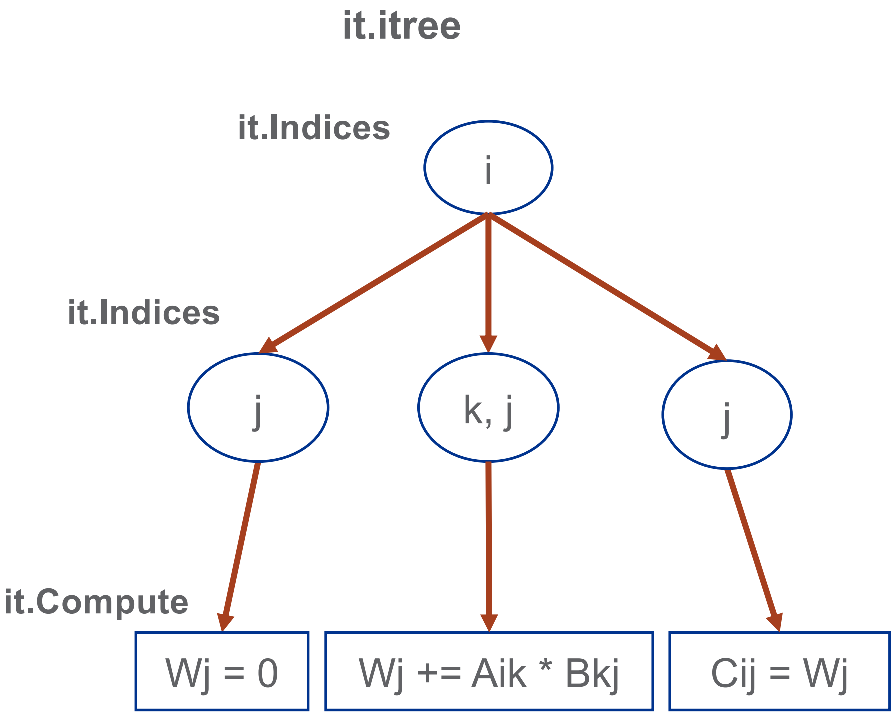
```
for i in I
  for j in J
    Wj = 0
  for k in K
    for j in J
      Wj += Aik * Bkj
  for j in J
    Cij = Wj
  // update pos/crd
```

Index tree for SpGEMM

Pseudo-code for SpGEMM with workspace

Index Tree IR Operations

- Three types of index tree IR operations
  - **it.itree**: the identifier of the index tree op in IT IR
  - **it.Indices**: represent the information in Index Node in index tree
  - **it.Compute**: represent the information in Compute Node in index tree



Index tree example

```
%96 = it.Compute (%cst_40, %95) {…} : …-> (i64)
%97 = it.Indices (%96) {indices = [2]} : (i64) -> i64
%98 = it.Compute (%34, %68, %95) {semiring="plus-times"} …: -> (i64)
%99 = it.Indices (%98) {indices = [1, 2]} : (i64) -> i64
%100 = it.Compute (%95, %93) {…} -> (i64)
%101 = it.Indices (%100) {indices = [2]}
%102 = it.Indices (%97, %99, %101) {indices = [0]}
%103 = it.itree (%102) : (i64) -> i64
```

Corresponding index tree IR

Generated Index Tree IR Operations Example

```
def main() {
    #IndexLabel Declarations
    IndexLabel [i] = [?];
    IndexLabel [j] = [?];
    IndexLabel [k] = [?];

    #Tensor Declarations
    Tensor<double> A([i, k], {CSR});
    Tensor<double> B([k, j], {CSR});
    Tensor<double> C([i, j], {CSR});

    #Tensor Data Initialization
    A[i, k] = comet_read(0);
    B[k, j] = comet_read(1);
    C[i, j] = 0.0;

    #Tensor Contraction
    C[i, j] = A[i, k] @(+,*) B[k, j];
}
```

SpGEMM DSL

```
%96 = it.Compute (%cst_40, %95) {…} : …-> (i64)
%97 = it.Indices (%96) {indices = [2]} : (i64) -> i64
%98 = it.Compute (%34, %68, %95) {semiring=“plus-times”} …: -> (i64)
%99 = it.Indices (%98) {indices = [1, 2]} : (i64) -> i64
%100 = it.Compute (%95, %93) {…} -> (i64)
%101 = it.Indices (%100) {indices = [2]}
%102 = it.Indices (%97, %99, %101) {indices = [0]}
%103 = it.itree (%102) : (i64) -> i64
```

SpGEMM Index Tree IR Ops

```
for i in I
  for j in J
    Wj = 0
  for k in K
    for j in J
      Wj += Aik * Bkj
  for j in J
    Cij = Wj
  // update pos/crd
```

Pseudo-code for SpGEMM with workspace

The semiring attribute influences how this code is generated.

Some semiring examples in DSL

```
def main() {
    #IndexLabel Declarations
    IndexLabel [a] = [?];
    IndexLabel [b] = [?];
    IndexLabel [c] = [?];

    #Tensor Declarations
    Tensor<double> A([a, b], {CSR});
    Tensor<double> B([b, c], {CSR});
    Tensor<double> C([a, c], {CSR});

    #Tensor Data Initialization
    A[a, b] = comet_read(0);
    B[b, c] = comet_read(1);

    #PlusTimes semiring
    C[a, c] = A[a, b] @(+,*) B[b, c];
}
```

```
def main() {
    #IndexLabel Declarations
    IndexLabel [a] = [?];
    IndexLabel [b] = [?];


    #Tensor Declarations
    Tensor<double> A([a, b], {CSR});
    Tensor<double> B([a, b], {CSR});

    #Tensor Data Initialization
    A[a, b] = comet_read(0);
    B[a, b] = comet_read(1);


    #Min monoid
    C[a, b] = A[a, b] @(min) B[a, b];
}
```

Semiring Operations in COMET

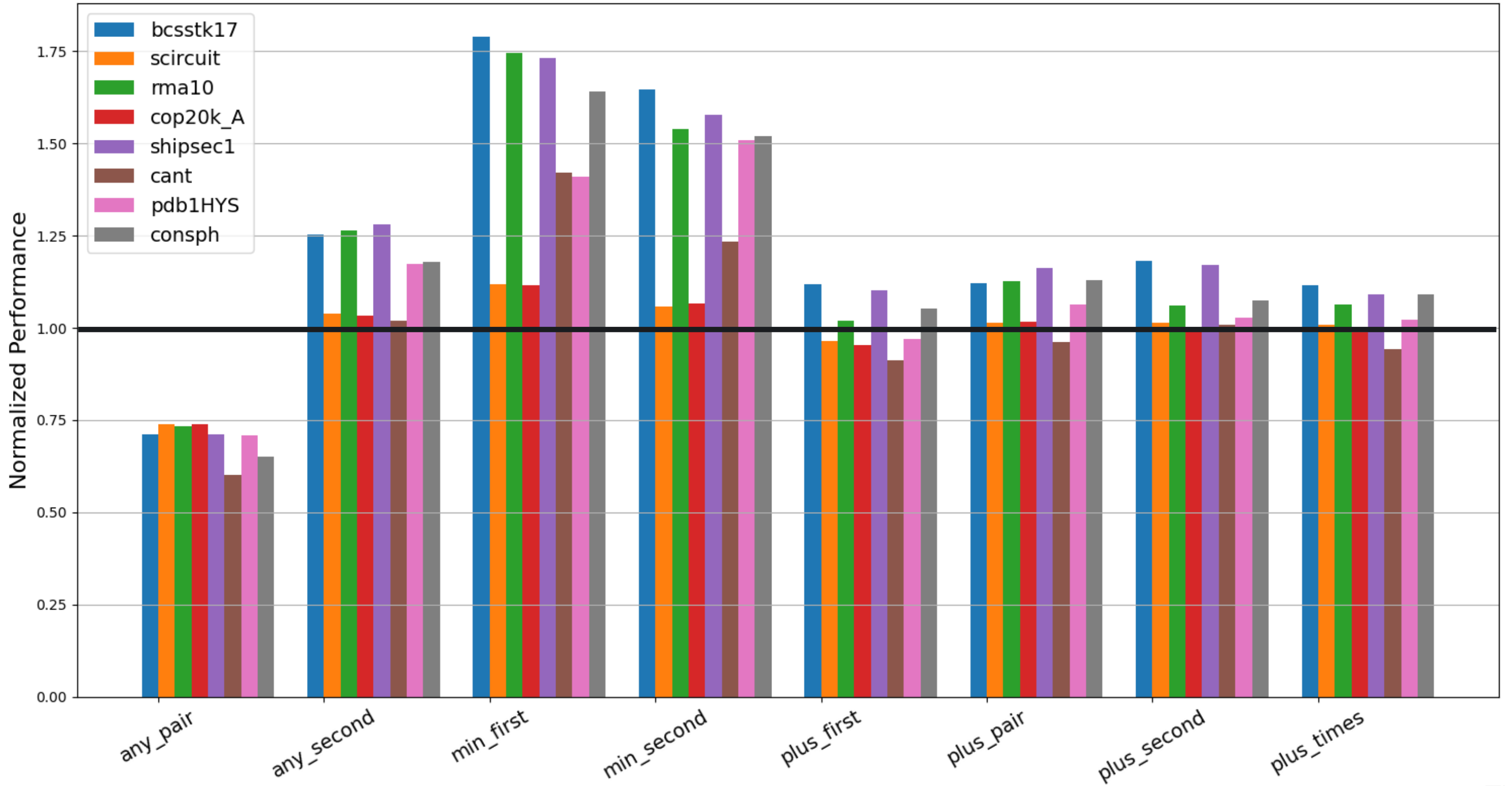
Semirings	Operation	Explanation
Lor-land	s(|, &)	‘lor’ means logical OR; ‘land’ means logical AND.
Min-first	s(min, first)	‘min’ means the minimal value; ‘first’ means first(x, y) = x: output the value of the first in the pair.
Plus-times	s(+,x)	‘+’ means addition; ‘x’ means multiplication.
Any-pair	s(any, pair)	‘any’ means “if there is any; if yes return true”. ‘pair’ means pair(x, y) = 1: x and y both have defined value at this intersection.
Min-plus	s(min, +)	‘min’ means the minimal value; ‘+’ means addition.
Plus-pair	s(+, pair)	‘+’ means addition; ‘pair’ means pair(x, y) = 1: x and y both have defined value at this intersection.
Min-second	s(min, second)	‘min’ means the minimal value; ‘second’ means second(x, y) = x: output the value of the second in the pair.
Plus-second	s(+, second)	‘+’ means addition; ‘second’ means second(x, y) = x: output the value of the second in the pair.
Plus-first	s(+, first)	‘+’ means addition; ‘first’ means first(x, y) = x: output the value of the first in the pair.

Semiring Operations per application

	Semirings	Operation	Description
BFS	Lor-land	s(|, &)	Compute traversal level for each vertex
	Min-first	s(min, first)	Compute parent for each vertex
	Plus-times	s(+,x)	Number of paths
	Any-pair	s(any, pair)	Reachability
	Min-plus	s(min, +)	Shortest path
SSSP	Min-plus	s(min, +)	Shortest path without mask (Bellman-Ford Algorithm)
TC	Plus-pair	s(+, pair)	Number of triangles
CC	Min-second	s(min, second)	Hooking and shortcutting
PR	Plus-second	s(+, second)	Outbound PageRank score
BC	Plus-first	s(+, first)	Accumulate path count

Semiring Performance (unjumbled)

- A method returns a matrix in an *unjumbled* state, with indices sorted
  - if the matrix will be immediately exported in unjumbled form, or
  - if the matrix is provided as input to a method that requires it to not be jumbled

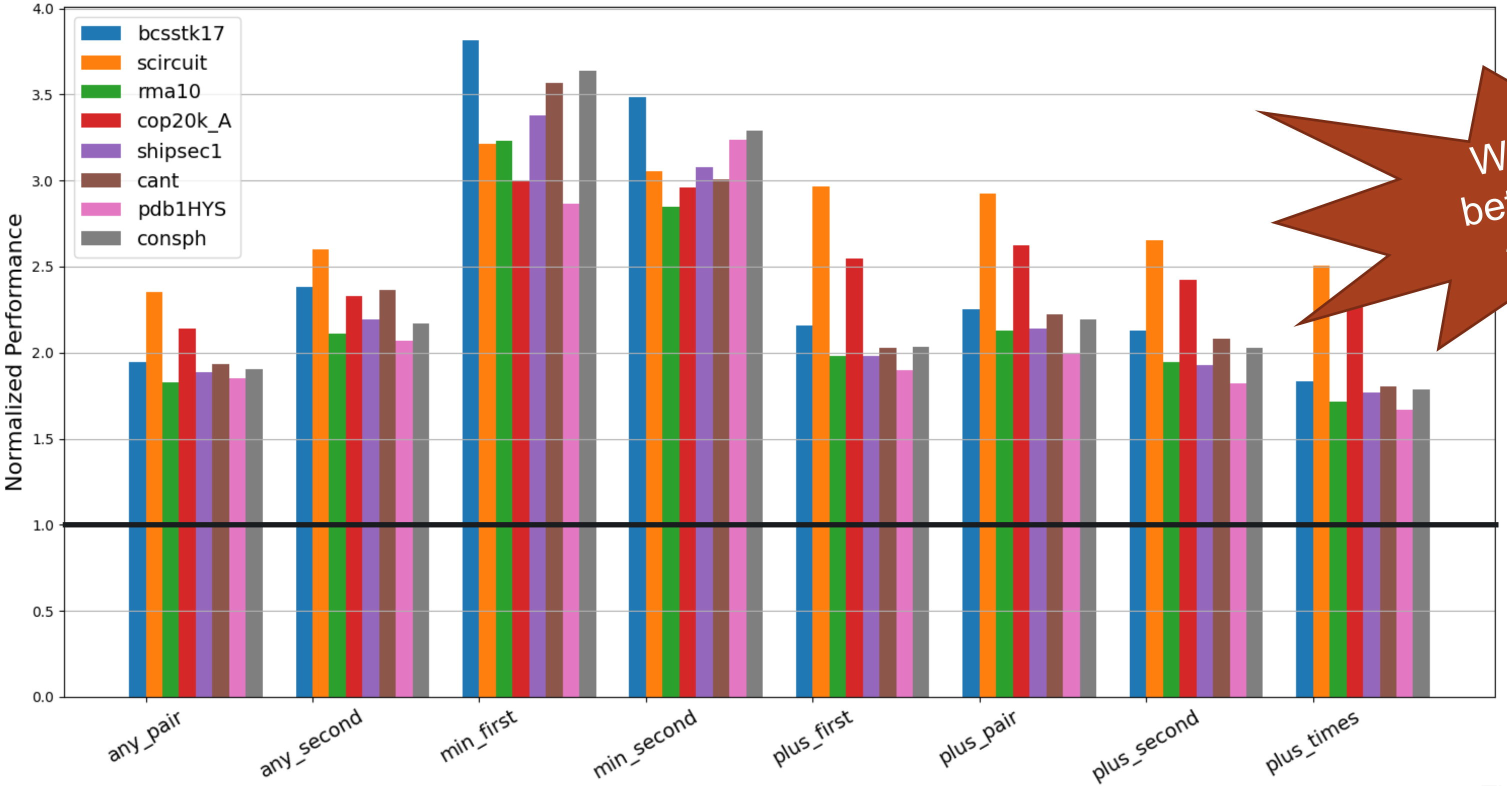


Performance comparison wrt LAGraph[1]

[1] Tim Mattson and others. “LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS”, IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019.

Semiring Performance (jumbled)

- A method returns a matrix in a *jumbled* state, with indices out of order
  - If some methods can tolerate jumbled matrices on input, the sorting of the indices is left pending

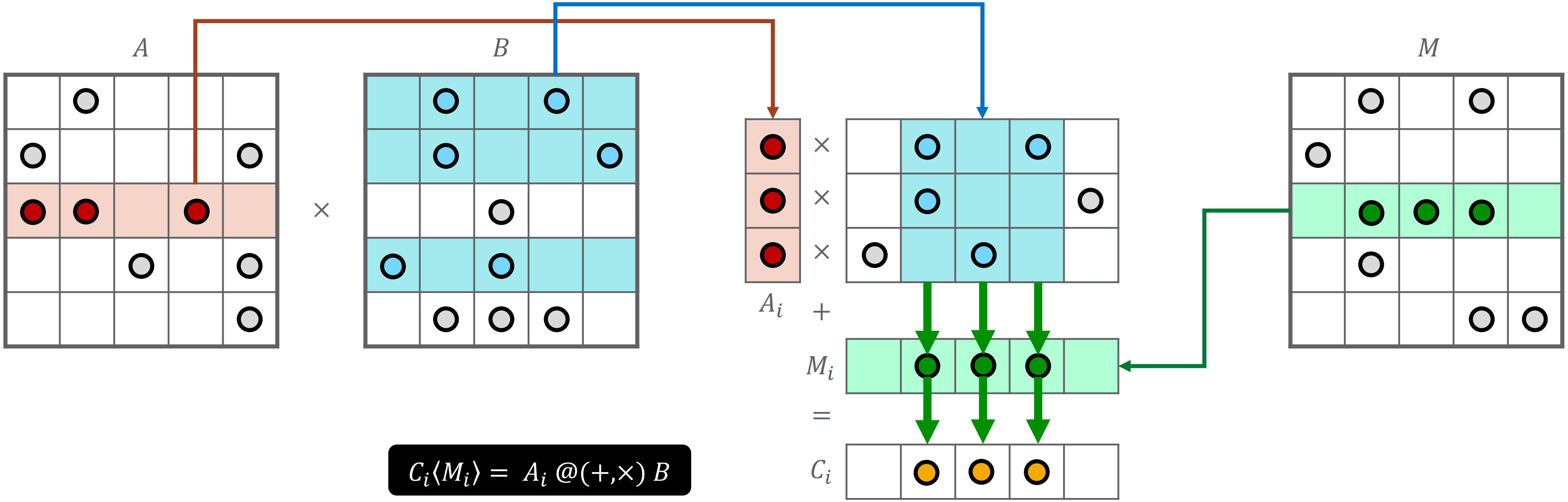


We need a better sorting algorithm

Performance comparison wrt LAGraph

Push-Based Masking

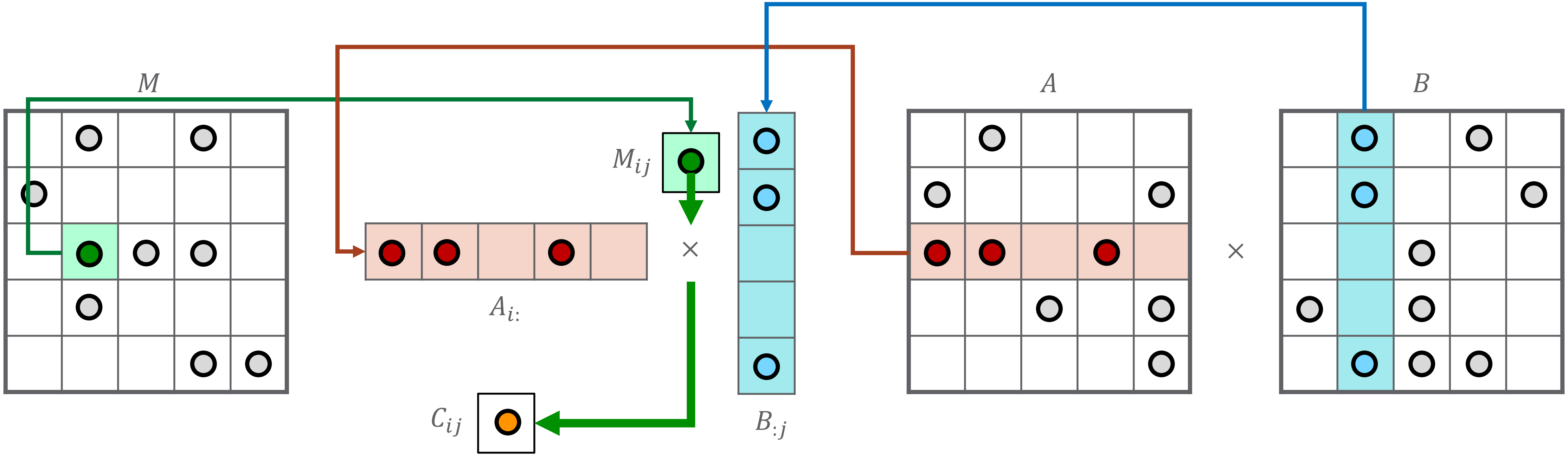
- Example of SpGEMM: $C\langle M\rangle = A\ @(+,\times)\ B$
- Driven by rows of A
- Do linear combination



$C_i\langle M_i\rangle = A_i\ @(+,\times)\ B$

Pull-Based Masking

- Example of SpGEMM: $C\langle M\rangle = A\ @(+,\times)\ B$
- Driven by non-zero elements of M
- Do dot product, and B is in CSC format

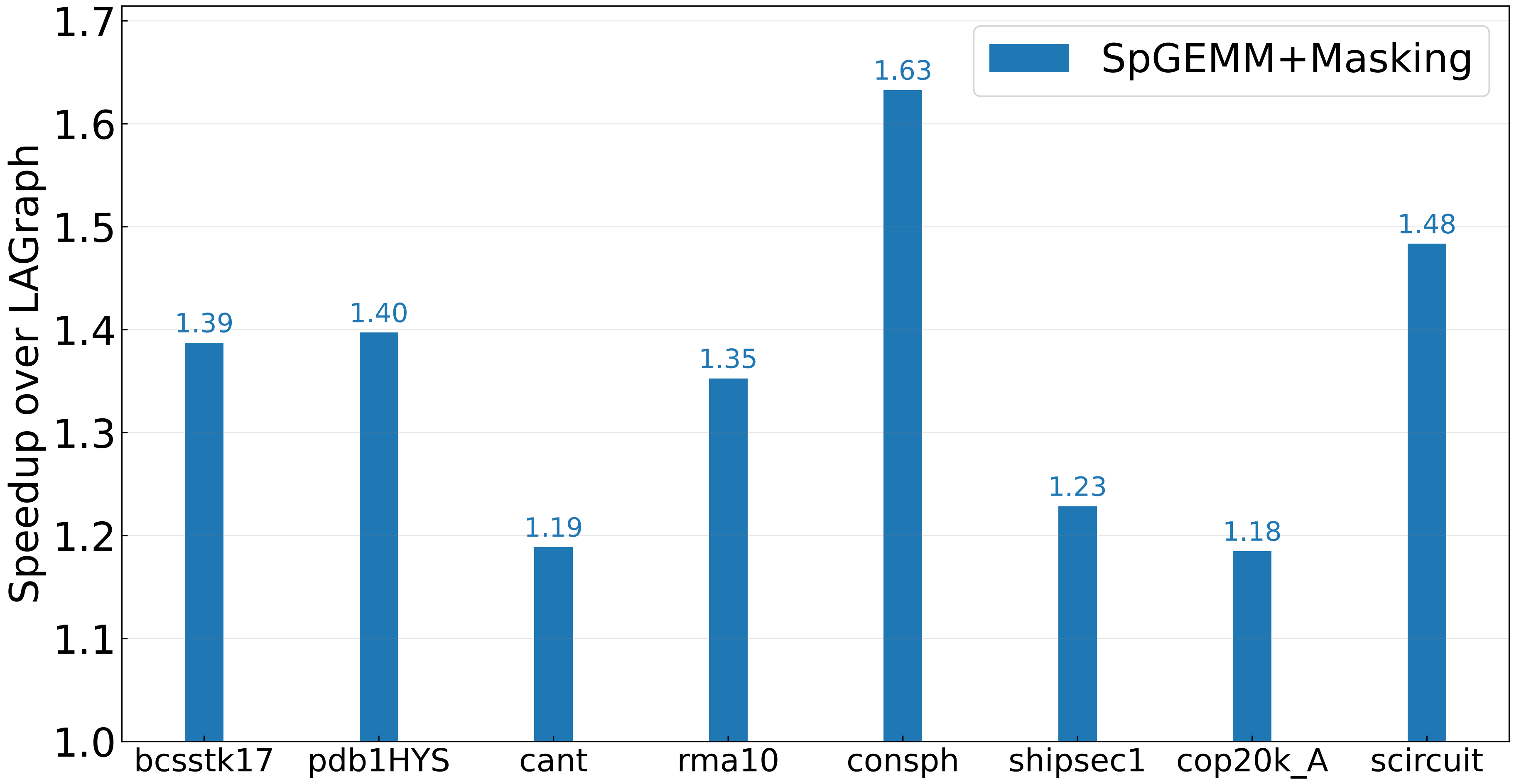


$$C_{ij}\langle M_{ij}\rangle = A_{i:}\ @(+,\times) B_{:j}$$

We plan to have pull-based masking in the future

PNNL-SA-182677

Results: Masking



Triangle Counting in COMET

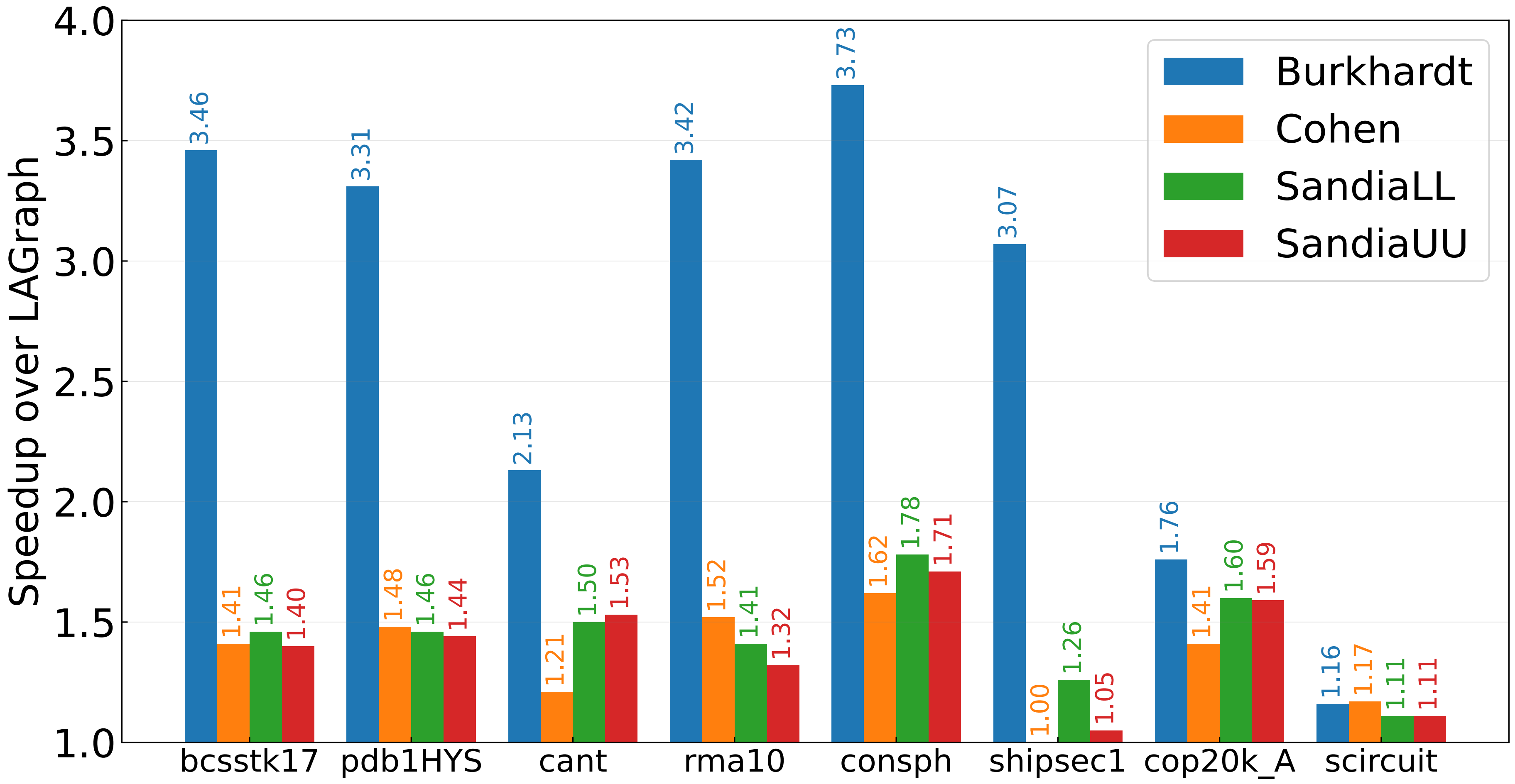
- Number of Triangles in a graph, where a triangle is a set of three mutually adjacent vertices in a graph.
- Various linear algebra-based algorithms proposed for the triangle counting problem.
    - Burkhard algorithm: $ntri = sum\big((A\ @(+,\times)A).*\ A\big)/6$
    - Cohen algorithm: $ntri = sum\big((L\ @(+,\times)U).*\ A\big)/2$
    - SandiaLL algorithm: $ntri = sum\big((L\ @(+,\times)L).*\ L\big)$
    - SandiaUU algorithm: $ntri = sum\big((U\ @(+,\times)U).*\ U\big)$

Triangle Counting in COMET

```
def main() {
    #IndexLabel Declarations
    IndexLabel [a] = [?];
    IndexLabel [b] = [?];
    IndexLabel [c] = [?];

    #Tensor Declarations
    Tensor<double> A([a, b], {CSR});
    Tensor<double> L([a, c], {CSR});
    Tensor<double> U([c, b], {CSR});

    #Tensor Data Initialization
    A[a, b] = comet_read(0, 1); # standard matrix read
    L[a, c] = comet_read(0, 2); # lower triangular read
    U[c, b] = comet_read(0, 4); # upper triangular read

    #PlusTimes semiring
    var ntri = SUM((L[a, c] @(+,*) U[c, b]) .* A[a, b])/2;
}
```

$$ntri = sum\big((L\ @(+,\times)U).*\ A\big)/2$$

Results: Triangle Counting



Conclusions and future work

- A DSL for implementing graph algorithms using linear algebra operations.
  - Support for semirings and masking.
- Optimizations for efficient codegen of sparse operations.
  - Workspace transforms.
- Sparse linear algebra operations as building blocks for graph algorithms paves the way for compiler optimizations.
  - Target heterogeneous accelerators.

Pacific Northwest
NATIONAL LABORATORY

Thank you

rizwan.ashraf@pnnl.gov
zhen.peng@pnnl.gov
lenny.guo@pnnl.gov
gokcen.kestor@pnnl.gov

Pacific
Northwest
NATIONAL LABORATORY

Acknowledgements

DMC
DATA-MODEL
CONVERGENCE
INITIATIVE
@PNNL

CO-DESIGN CENTER FOR
ARTIFICIAL INTELLIGENCE-FOCUSED ARCHITECTURES AND ALGORITHMS
(ARIAA)

- The research described in this presentation is part of the Data Model Convergence Initiative at Pacific Northwest National Laboratory (PNNL). It was conducted under the Laboratory Directed Research and Development Program at PNNL, a multi-program national laboratory operated by Battelle for the U.S. Department of Energy (DOE).
- The research described in this presentation is also supported in part through U.S. Department of Energy’s Office of Advanced Scientific Computing Research as part of the Center for Artificial Intelligence-focused Architectures and Algorithms (ARIAA).
- PNNL is operated by Battelle for the DOE under Contract DE-AC05-76RL01830.