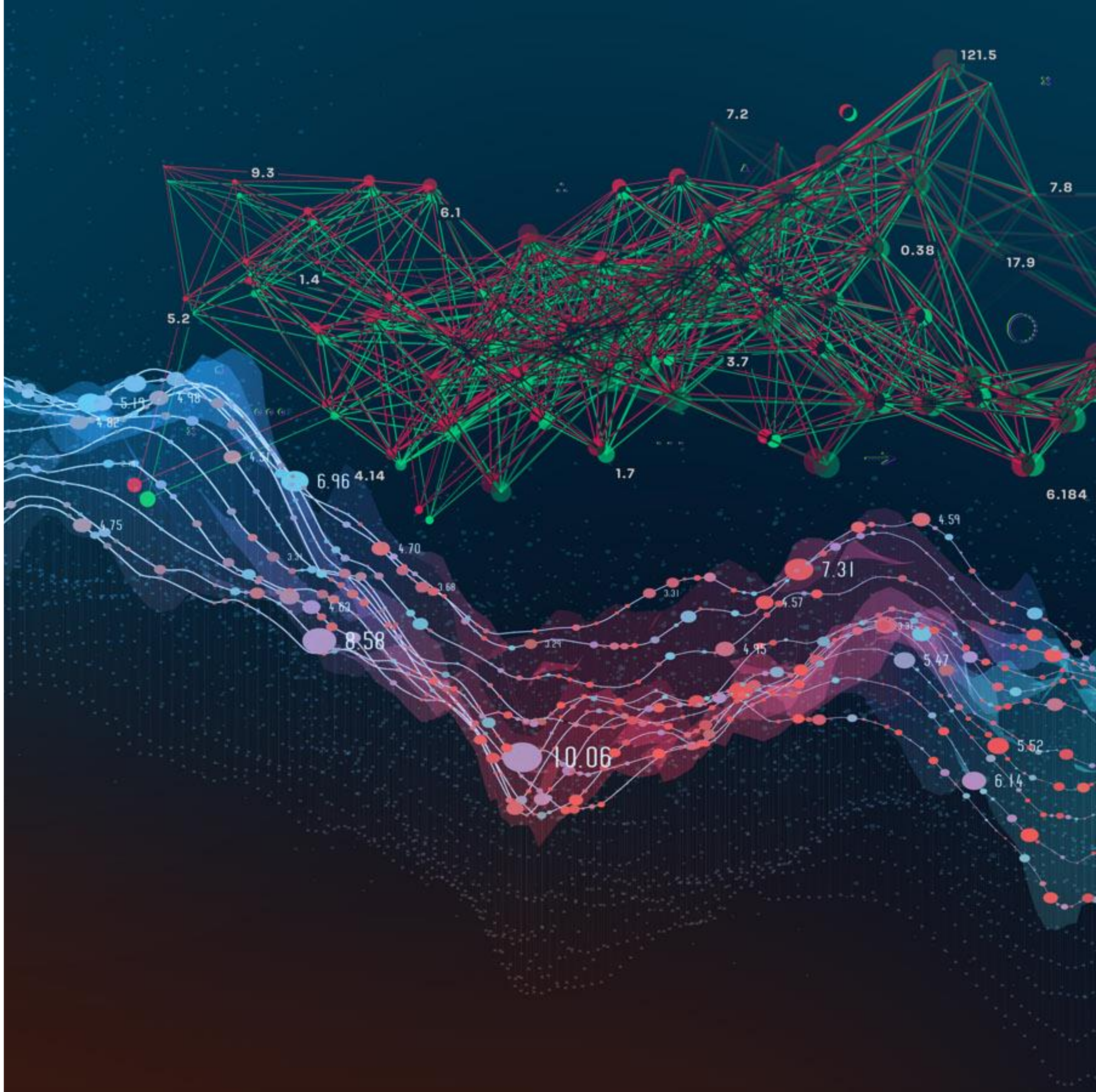
Sparse Tensor Algebra Optimizations in MLIR

November 2021

Ruiqin Tian, Luanzheng Guo, Gokcen Kestor
Pacific Northwest National Lab

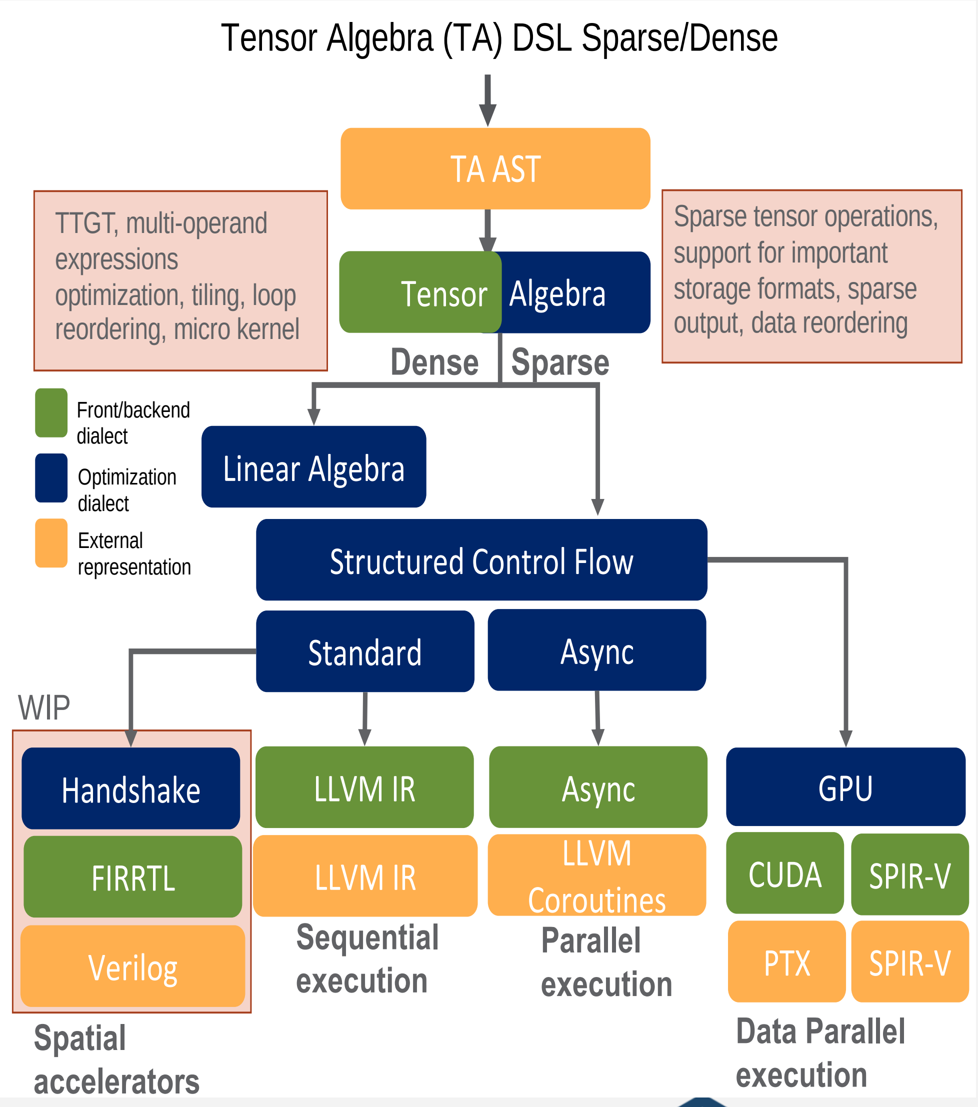
PNNL is operated by Battelle for the U.S. Department of Energy



COMET[1,2]: Domain specific Compilation in Multi-Level IR

- COMET Domain specific language (DSL)
  - Domain specific language for **sparse** and **dense** tensor algebra, focusing on computational chemistry and graph analytics applications
- COMET compiler infrastructure
  - Enable from high-level, domain-specific and low-level, architecture-specific compiler optimizations
    - ✓ **Multi-level** code optimizations, including domain and architecture specific
  - **Tensor algebra dialect** in the MLIR infrastructure
  - **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** from mixed sparse-dense computation
  - Automatic code generation for **sequential** and **parallel** execution
  - 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

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



[1] COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry. Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor, The 33rd the Workshop on Languages and Compilers for Parallel Computing (LCPC), October, 2020.
[2] A High Performance Sparse Tensor Algebra Compiler in MLIR. Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, Gokcen Kestor. The Seventh Annual Workshop on the LLVM Compiler Infrastructure in HPC, November 2021.

Motivation

Sparse tensor algebra is widely used in many applications, including scientific computing, machine learning, and data analytics.

In sparse kernels, both input tensors might be sparse, and generates sparse output tensor.

Challenges

- Sparse tensors are stored in compressed irregular data structure, which introduces irregular data access pattern and affect data locality.
- Compound expression, i.e., the output of an operation will be used as input in other operations.
- Storing output tensor in dense format is not an option due to memory overhead.
- Sparse output contains expensive insertions and accesses to sparse tensors, which has large time complexity

Pacific Northwest NATIONAL LABORATORY

Our solution

- 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

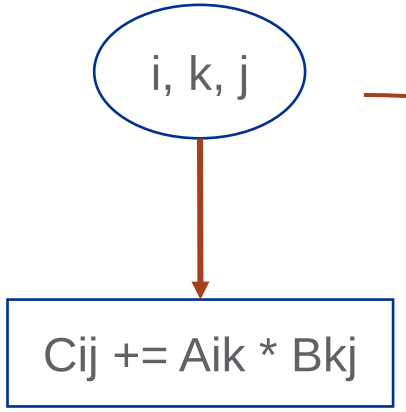
[1] Tensor Algebra Compilation with Workspaces. Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, Saman Amarasinghe , Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, 2019

Pacific Northwest
NATIONAL LABORATORY

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

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



Index tree for SpGEMM

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

Pseudo-code for SpGEMM

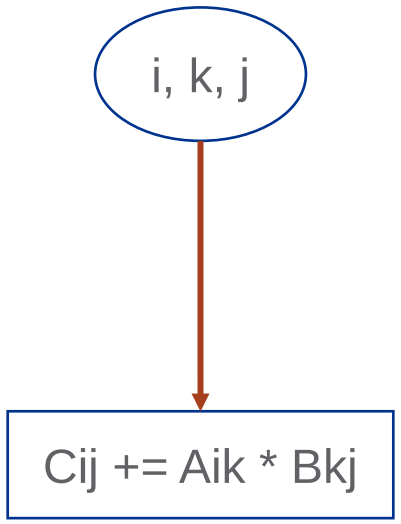
Pacific Northwest
NATIONAL LABORATORY

Workspace Transformation

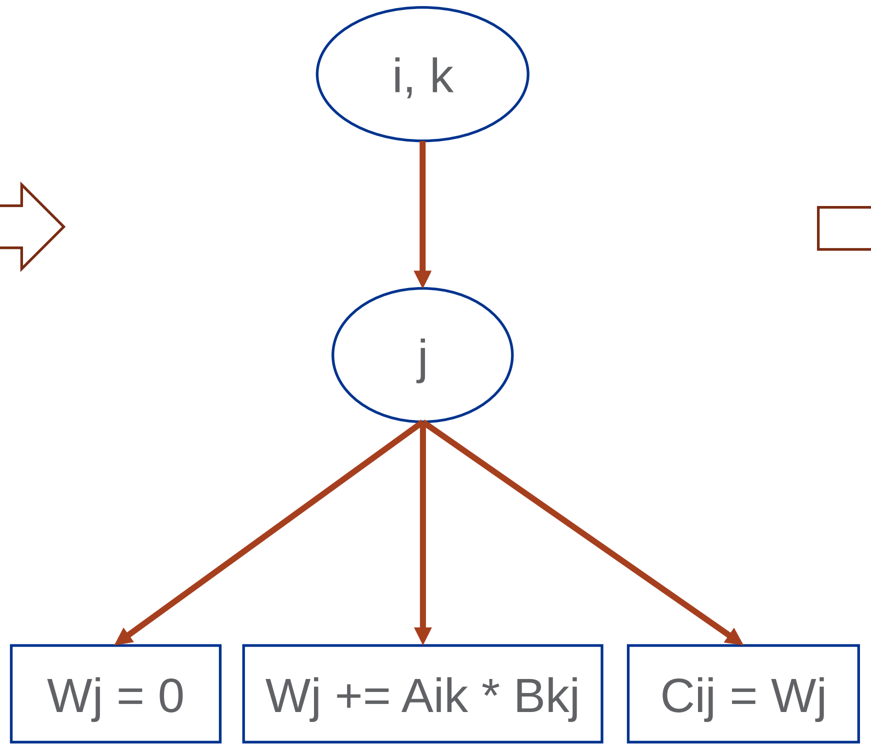
- We perform compiler transformation in the index tree representation of a tensor expression
  - Benefits
    - ✓ Reduces expensive insertions/ accesses to sparse tensors
      - Dense data structure has better locality
      - Generates “for” loops instead of “while” loops
      - Utilize the existing 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?

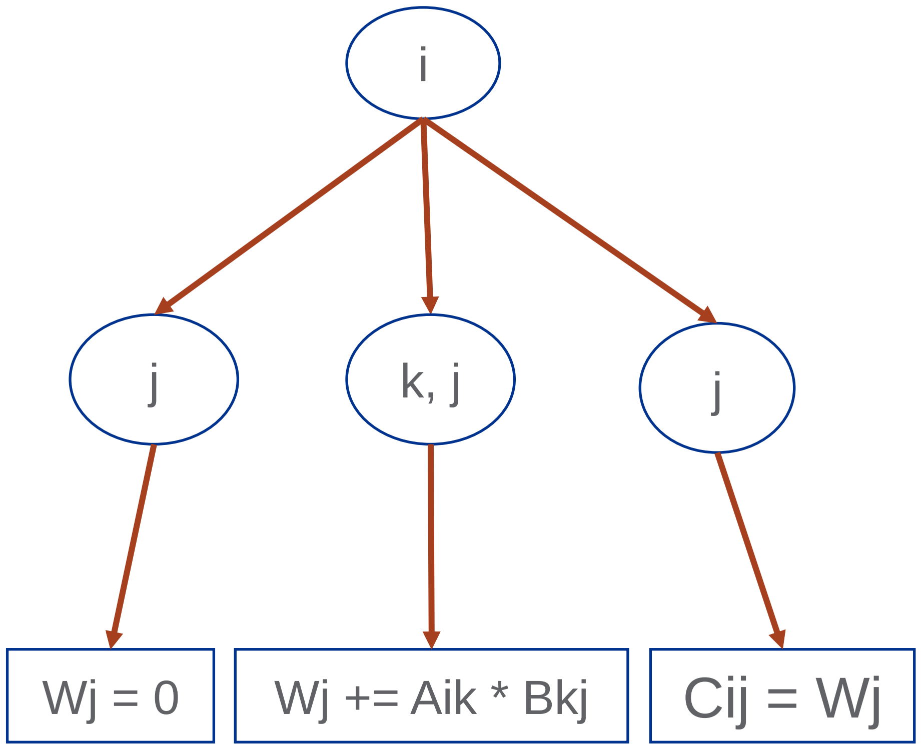
Transformation in Index Tree



Index tree for SpGEMM

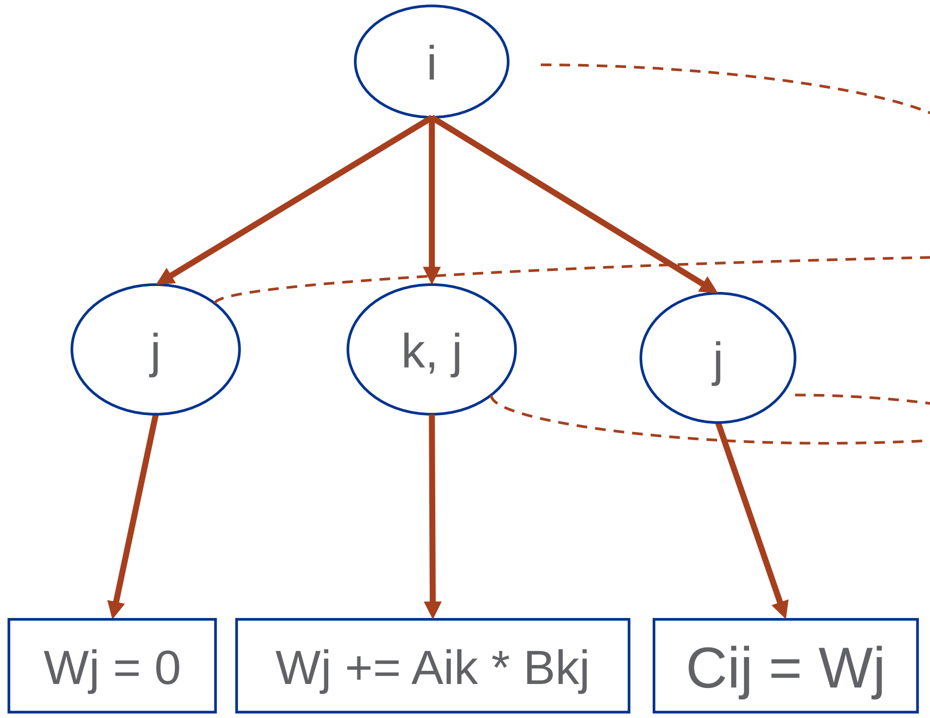


Index tree for SpGEMM with workspace



Reduce unnecessary loops

Code Generation from Index Tree IR Operations



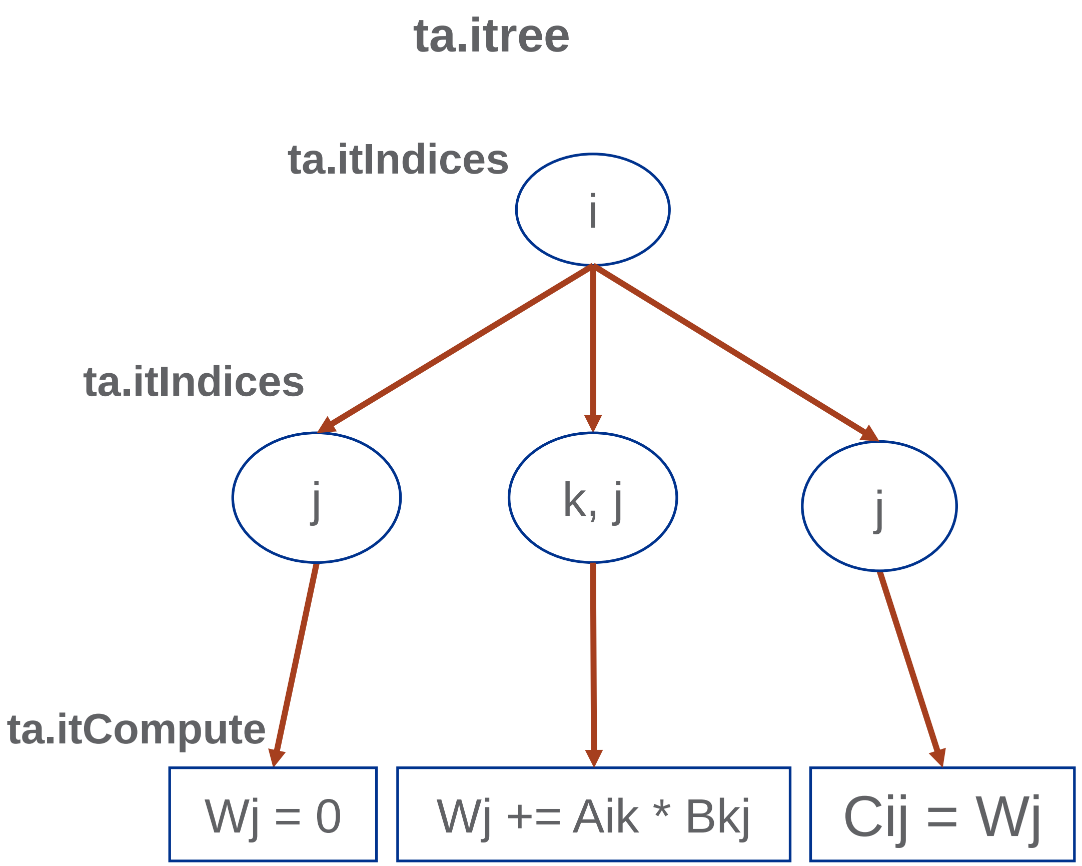
Index tree for SpGEMM

```
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

Index Tree IR Operations

- Three types of index tree IR operations
  - **ta.itree**: the identifier of the index tree op in TA IR
  - **ta.itIndices**: represent the information in Index Node in index tree
  - **ta.itCompute**: represent the information in Compute Node in index tree



Index tree example

```
    %96 = ta.itCompute(%cst_40, %95) {allFormats = [[], ["D"]], allPerms =
[[], [2]], op_type = 0 : i64} : (f64, tensor<?xf64>) -> (i64)
    %97 = "ta.itIndices"(%96) {indices = [2]} : (i64) -> i64
    %98 = ta.itCompute(%34, %68, %95) {allFormats = [["D", "CU"], ["D",
"CU"], ["D"]], allPerms = [[0, 1], [1, 2], [2]], op_type = 2 : i64} :
(!ta.sptensor<tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xf64>, index, index, index,
index, index, index, index>, !ta.sptensor<tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xi32>,
tensor<?xf64>, index, index, index, index, index, index, index>, tensor<?xf64>) -> (i64)
    %99 = "ta.itIndices"(%98) {indices = [1, 2]} : (i64) -> i64
    %100 = ta.itCompute(%95, %93) {allFormats = [["D"], ["D", "CU"]],
allPerms = [[2], [0, 2]], op_type = 0 : i64} : (tensor<?xf64>,
!ta.sptensor<tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xf64>, index, index, index,
index, index, index, index>) -> (i64)
    %101 = "ta.itIndices"(%100) {indices = [2]} : (i64) -> i64
    %102 = "ta.itIndices"(%97, %99, %101) {indices = [0]} : (i64, i64, i64)
-> i64
    %103 = "ta.itree"(%102) : (i64) -> i64
```

Corresponding index tree IR

Generated Index Tree IR Operations Example

$$C_{ij} = \sum_k A_{ik} * B_{kj}$$

```
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();
    B[k, j] = comet_read();
    C[i, j] = 0.0;

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

SpGEMM DSL

```
    %96 = ta.itCompute(%cst_40, %95) {allFormats = [[], ["D"]], allPerms =
[[], [2]], op_type = 0 : i64} : (f64, tensor<?xf64>) -> (i64)
    %97 = "ta.itIndices"(%96) {indices = [2]} : (i64) -> i64
    %98 = ta.itCompute(%34, %68, %95) {allFormats = [["D", "CU"], ["D",
"CU"], ["D"]], allPerms = [[0, 1], [1, 2], [2]], op_type = 2 : i64} :
(!ta.sptensor<tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xf64>, index, index, index,
index, index, index, index>, !ta.sptensor<tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xi32>,
tensor<?xf64>, index, index, index, index, index, index, index>, tensor<?xf64>) -> (i64)
    %99 = "ta.itIndices"(%98) {indices = [1, 2]} : (i64) -> i64
    %100 = ta.itCompute(%95, %93) {allFormats = [["D"], ["D", "CU"]],
allPerms = [[2], [0, 2]], op_type = 0 : i64} : (tensor<?xf64>,
!ta.sptensor<tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xf64>, index, index, index,
index, index, index, index>) -> (i64)
    %101 = "ta.itIndices"(%100) {indices = [2]} : (i64) -> i64
    %102 = "ta.itIndices"(%97, %99, %101) {indices = [0]} : (i64, i64, i64)
-> i64
    %103 = "ta.itree"(%102) : (i64) -> i64
```

SpGEMM Index Tree IR Ops

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();
    B[k, j] = comet_read();
    C[i, j] = 0.0;

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

SpGEMM DSL

```
    %96 = ta.itCompute(%cst_40, %95) {allFormats = [[], ["D"]], allPerms = [[], [2]],
op_type = 0 : i64} : (f64, tensor<?xf64>) -> (i64)
    %97 = "ta.itIndices"(%96) {indices = [2]} : (i64) -> i64
    %98 = ta.itCompute(%34, %68, %95) {allFormats = [["D", "CU"], ["D", "CU"], ["D"]],
allPerms = [[0, 1], [1, 2], [2]], op_type = 2 : i64} : (!ta.sptensor<tensor<?xi32>,
tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xf64>, index, index, index, index,
index, index, index>, !ta.sptensor<tensor<?xi32>, tensor<?xi32>, tensor<?xi32>,
tensor<?xi32>, tensor<?xf64>, index, index, index, index, index, index, index>,
tensor<?xf64>) -> (i64)
    %99 = "ta.itIndices"(%98) {indices = [1, 2]} : (i64) -> i64
    %100 = ta.itCompute(%95, %93) {allFormats = [["D"], ["D", "CU"]], allPerms = [[2],
[0, 2]], op_type = 0 : i64} : (tensor<?xf64>, !ta.sptensor<tensor<?xi32>, tensor<?xi32>,
tensor<?xi32>, tensor<?xi32>, tensor<?xf64>, index, index, index, index, index, index,
index>) -> (i64)
    %101 = "ta.itIndices"(%100) {indices = [2]} : (i64) -> i64
    %102 = "ta.itIndices"(%97, %99, %101) {indices = [0]} : (i64, i64, i64) -> i64
  %103 = "ta.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

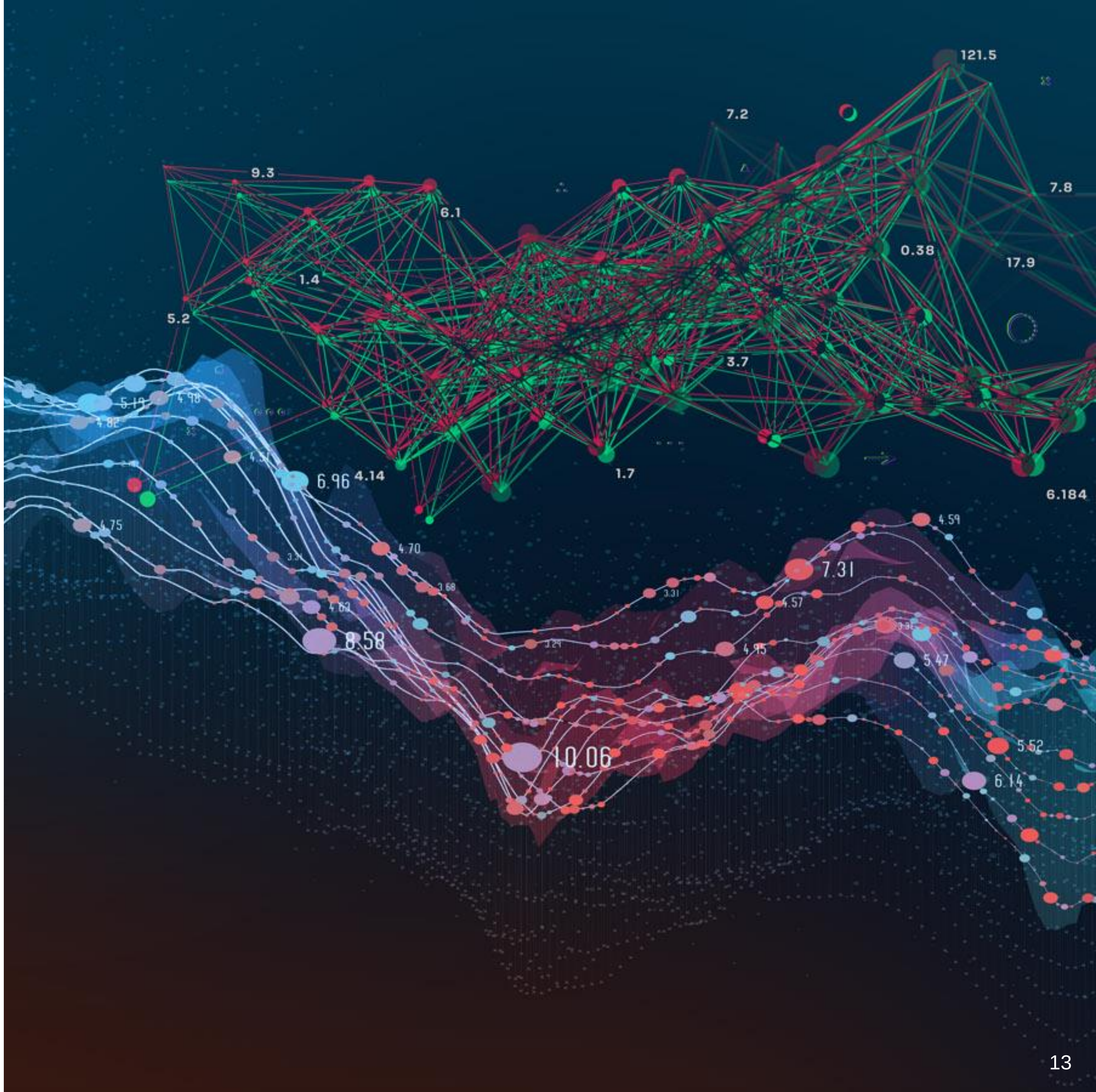
Conclusion

- Targeting sparse tensor algebra computations
  - Sparse inputs and sparse output
- Introduce "workspace"
- Workspace transformation
- Index tree IR operations

Pacific Northwest
NATIONAL LABORATORY

Thank you

ruiqin.tian@pnnl.gov
gokcen.kestor@pnnl.gov



Evaluation

Next steps

- Additional sparse linear algebra operators in the COMET DSL and COMET compiler.
- Extend the Rust eDSL to support 1) the new COMET operators and 2) more efficient integration. We will also implement initial versions of triangle counting and evaluate programmability and performance.
- Evaluate the performance of automatically generated graph algorithms by COMET on CPU and GPU architectures.
- Explore possible strategies for lowering of COMET tensor algebra dialect to IR representations that can be executed on SambaNova SN10 and Xilinx Versal AIE.

Tensor Algebra Domain-specific Language

Tensor algebra domain-specific language to express tensor operations

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

A is CSR format
B and C are dense format

```
def main() {
    #IndexLabel Declarations
    IndexLabel [i] = [?];
    IndexLabel [j] = [?];
    IndexLabel [k] = [32];

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

    #Tensor Data Initialization
    A[i, k] = comet_read();
    B[k, j] = 1.0;
    C[i, j] = 0.0;

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

Support for dynamic tensors' sizes

Support for common sparse formats

Runtime utility functions

Familiar Einstein notation

COMET compiler

- COMET compiler infrastructure
  - Internal tensor storage format
  - Sparse Tensor Algebra dialect, including a sparse storage data type and sparse tensor contraction operation
  - Code generation to support a wide range of sparse storage formats
  - Parallel execution
- COMET runtime utility functions
  - Read Matrix Market format, convert it into internal storage format
- Data layout optimizations
  - Optimize the memory access pattern for better performance