Pacific Northwest
NATIONAL LABORATORY

COMET: Domain Specific COMpiler for Extreme Targets in Multi-Level IR

October 25, 2022

Gokcen Kestor, Rizwan Ashraf, Zhen Peng,

Ruiqin Tian, Luanzheng Guo, Ryan Friese

Pacific Northwest National Laboratory

U.S. DEPARTMENT OF ENERGY BATTELLE

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

Pacific Northwest
NATIONAL LABORATORY

COMET Compiler Tutorial

- **Objectives**: This tutorial will provide an overview of the COMET compiler and its supported frontends. It will also review the optimizations and transformations supported by the COMET compiler for efficient code generation on target architectures. At the end of this tutorial, attendees should learn how to write various dense and sparse tensor algebra kernels using the COMET DSL, perform sophisticated code optimizations and transformations.

- **What you will learn**:
  - Write sparse and dense tensor algebra kernels in COMET DSL
  - Apply various code optimizations and transformations for high performance
  - FPGA code generation flow

Additional Information

- Tutorial web page:
  - **https://cometcompiler.github.io/tutorials/pact22/pact22.html**
- Breaks:
  - We will follow the timeframe for breaks provided by the PACT program. Only one break between 10.30-11.00 (CST)
- COMET source code and documentation:
  - GIT repository: **https://github.com/pnnl/COMET**
  - Documentation: **https://pnnl-comet.readthedocs.io**
- Docker container for hands-on sessions

```
docker pull gkestor/duomo:latest
docker run -it gkestor/duomo /bin/bash
```

Pacific Northwest NATIONAL LABORATORY

Agenda

Time (CDT)	Topic	Presenter	Duration
9.00-9.10	Logistic	G. Kestor	10 minutes
9.10-9.50	Dense and Sparse Tensor Algebra	G. Kestor	40 minutes
9.50-10.30	Hands-on Session	G. Kestor	40 minutes
10.30-11.00	Break		
11.00-11.10	Kernel Fusion	G. Kestor	10 minutes
11.10-11.20	Semiring Support	R. Ashraf	10 minutes
11.20-11.35	FPGA Codegen	R. Ashraf	15 minutes
11.35-12.15	Hands-on Session	R. Ashraf	40 minutes
12.15-12.30	Conclusions and Q&A	All	15 minutes

Team

Gokcen Kestor
PNNL

Rizwan Ashraf
PNNL

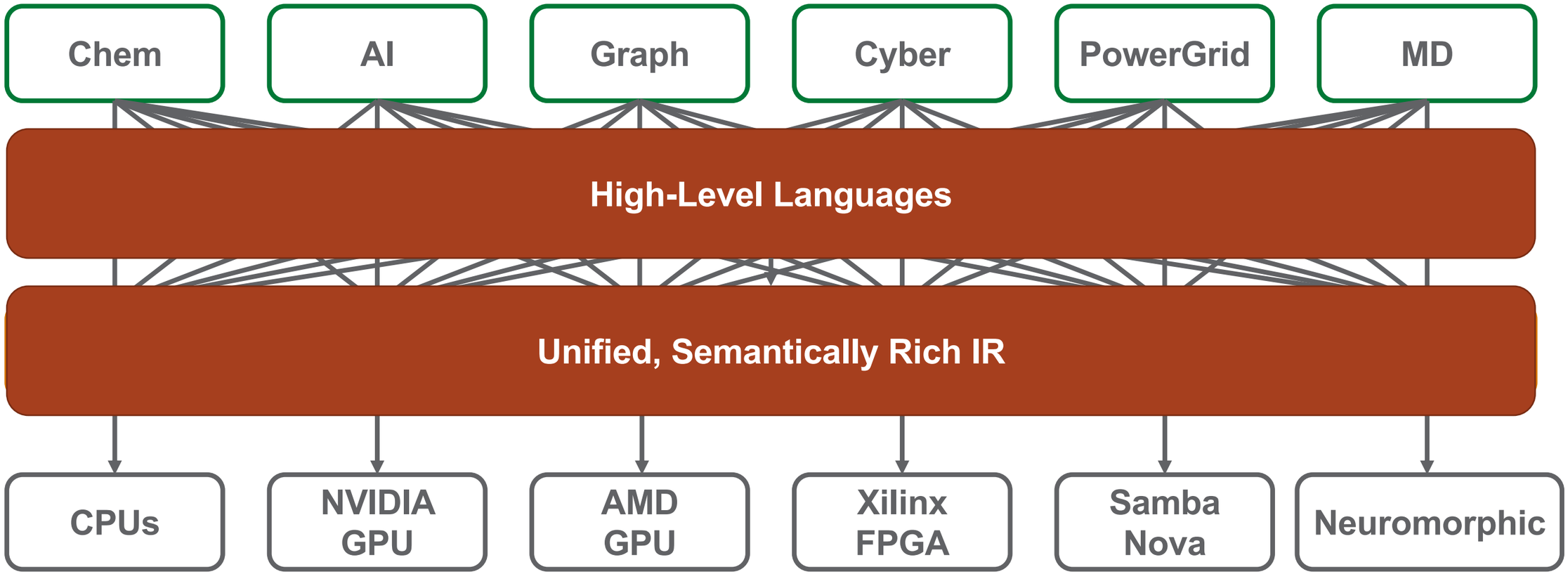
Ryan Friese
PNNL

Lenny Guo
PNNL

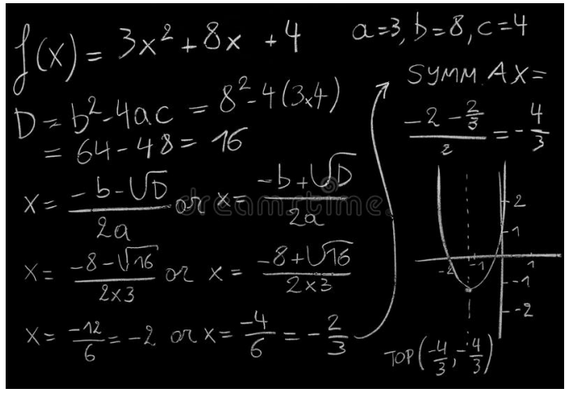
Pacific Northwest NATIONAL LABORATORY

Programming Heterogeneous Systems

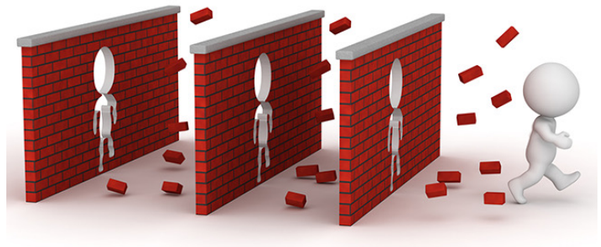
- Porting applications to heterogeneous systems often requires porting code to different programming environments
  - Explosion of complexity and versioning
  - Difficult to achieve performance portability

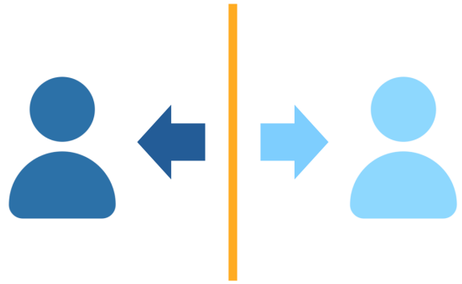


Vision: A Unified, Semantically Rich IR



- To achieve performance portability, it is paramount to identify computational patterns/motifs
  - High-level languages allow users to express high-level computational patterns/motifs
  - Semantics information is used for efficient code generation
- Remove artificial barriers that hinder progress
  - Share code and architectural optimizations
  - Re-use compiler passes, optimization and codegen algorithms
- Clear separation of responsibilities
  - Users implement algorithms using high-productive programming environments
  - Compiler generates efficient code for heterogeneous architectures

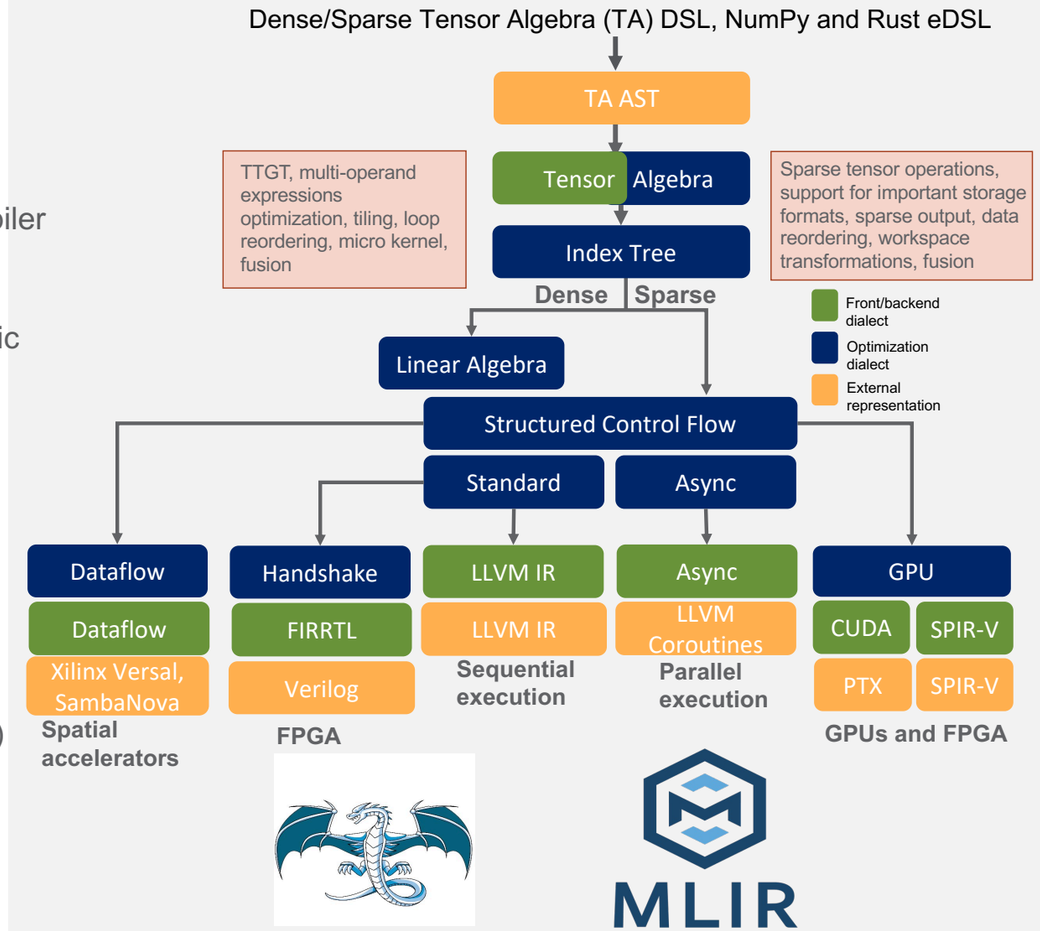




COMET: Domain Specific Compilation in Multi-level IR

- 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

MLIR

New compiler infrastructure

Part of LLVM project

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

Pacific Northwest NATIONAL LABORATORY

COMET High-Level Programming Abstraction

```
def main() {
        #IndexLabel Declarations
        IndexLabel [i] = [?];
        IndexLabel [k] = [?];
        IndexLabel [j] = [312];

        #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(filename)
        B[k, j] = 1.0;
        C[i, j] = 0.0;

        #Sparse Matrix Dense Matrix Multiplication
        C[i, j] = A[i, k] * B[k, j];
}
```

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

Static and dynamic Index labels

User-specified tensor storage formats via either common name (e.g., CSR) or attribute per dimension

Runtime utility functions to read input matrices and tensors

Tensor operations (e.g., SpMM, SpMV, SpGEMM, element-wise multiplication, transpose)

Other supported COMET Frontends

COMET DSL

```
def main() {
        #IndexLabel Declarations
        IndexLabel [a] = [?];
        IndexLabel [b] = [312];
        IndexLabel [c] = [?];

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

        #Tensor Data Initialization
        A[a, c] = comet_read(filename)
        B[c, b] = 1.0;
        C[a, b] = 0.0;

        #Sparse Matrix Dense Matrix Multiplication
        C[a, b] = A[a, c] * B[c, b];
}
```

- Development and testing
- High-level representation
- Resembles mathematical notations

NumPy

```
1   import numpy as np
2   import comet
3
4   def compute_einsum():
5       A = np.ones([3,3], dtype=float)
6       B = np.full([3,3], 2, dtype=float)
7       C = comet.einsum('ij,jk->ik',A,B)
8       return C
9
10  compute_einsum()
```

- Drop-in replacement for NumPy code
- Extend NumPy with non-standard operators

Rust eDSL

```
1   fn main() {
2       let a = Index::new();
3       let b = Index::new();
4       let c = Index::with_value(32);
5
6       let A = Tensor::<f64>::csr(vec![&a, &b]).fill_from_file("file.txt");
7       let B = Tensor::<f64>::csr(vec![&b, &c]).fill(1.0);
8       let mut C = Tensor::<f64>::dense(vec![&a, &c]).fill(0.0);
9       let mut D = Tensor::<f64>::dense(vec![&b, &a]).fill(0.0);
10
11      eval!{
12          C = &A * &B;
13          f = A.sum();
14          D = A.transpose();
15      };
16  }
```

- Full-fledge, high-performance language
- Enables Lamellar[1] distributed runtime
- Add support for heterogeneous computing to Rust
- Integrate with large number of external packages

[1] Lamellar - Rust HPC runtime, https://github.com/pnnl/lamellar

Pacific Northwest
NATIONAL LABORATORY

COMET Dense Tensor Algebra

Computational Chemistry: Dense Tensor Contractions

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
   for (k=0; k<N; k++)
     for (l=0; l<N; l++)
      for (m=0; m<N; m++)
        for (n=0; n<N; n++)
           C[i][j][k][l] = A[i][m][k][n]*B[j][l][m][n];
```

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

- NWChem is a high-performance computational chemistry suite
  - Includes dominant dense tensor contraction kernels, spends a significant fraction of supercomputer time
- Tensor contractions are higher dimensional analogs of matrix-matrix multiplication.
  - Naive loop-nest implementation results in ***poor performance due to lack of locality***
  - ***Enormous space of loop permutation/tiling + tile size*** selection for better performance

Pacific Northwest NATIONAL LABORATORY

Transpose-Transpose-GEMM-Transpose (TTGT)

- Reformulate a tensor contraction operation via TTGT for better performance efficiency
  - Most common approach used to perform tensor contractions to leverage optimized GEMM kernels
  - “Flatten” the tensors to matrices
  - Use GEMM (matrix-matrix multiplication) for contraction
  - “Unflatten” output matrix to tensor

$$C_{ablj} = \sum_{k} A_{bka} \cdot B_{jkl}$$

$$\tilde{A}_{(ba)k} \leftarrow A_{bka}$$
$$\tilde{B}_{k(jl)} \leftarrow B_{jkl}$$
$$GEMM(b*a, j*l, k, \tilde{A}, \tilde{B}, \tilde{C})$$
$$C_{ablj} \leftarrow \tilde{C}_{bajl}$$

Tensor Algebra Dialect (IR)

```
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];
}
```

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

```
#map0 = affine_map<(d0, d1, d2) -> (d0, d1)>
#map1 = affine_map<(d0, d1, d2) -> (d1, d2)>
#map2 = affine_map<(d0, d1, d2) -> (d0, d2)>


func @main() {
    %c0 = constant 0 : index
    %c1 = constant 1 : index
    %c32 = constant 32 : index
    %0 = "ta.index_label_dynamic"(%c0, %c1) : (index, index) -> !ta.range
    %1 = "ta.index_label_dynamic"(%c0, %c1) : (index, index) -> !ta.range
    %2 = "ta.index_label_static"(%c0, %c32, %c1) : (index, index, index) -> !ta.range

    %4 = "ta.sparse_tensor_decl"(%0, %1) {format = "CSR"} : (!ta.range, !ta.range) ->
            tensor<?x?xf64>
    %5 = "ta.dense_tensor_decl"(%1, %2) {format = "Dense"} : (!ta.range, !ta.range) ->
            tensor<?x32xf64>
    %6 = "ta.dense_tensor_decl"(%0, %2) {format = "Dense"} : (!ta.range, !ta.range) ->
            tensor<?x32xf64>

    "ta.fill_from_file"(%4) {filename = “filename.mtx”} : (tensor<?x?xf64>) -> ()
    "ta.fill"(%5) {value = 1.000000e+00 : f64} : (tensor<?x32xf64>) -> ()
    "ta.fill"(%6) {value = 0.000000e+00 : f64} : (tensor<?x32xxf64>) -> ()

    "ta.tc"(%4, %5, %6) {__alpha__ = 1.000000e+00 : f64, __beta__ = 0.000000e+00 :    f64,
formats = ["CSR", "Dense", "Dense"], indexing_maps = [#map0,  #map1, #map2]} :
(tensor<?x?xf64>, tensor<?x32xf64>,  tensor<?x32xf64>) -> ()

    "ta.return"() : () -> ()
}
```

Reformulation of Tensor contraction via TTGT

```
#map0 = affine_map<(d0, d1, d2, d3) -> (d2, d0, d3)>
#map1 = affine_map<(d0, d1, d2, d3) -> (d3, d2, d1)>
#map2 = affine_map<(d0, d1, d2, d3) -> (d1, d0)>



"ta.tc"(%4, %5, %6) {
              __alpha__ = 1.000000e+00 : f64,
              __beta__ = 0.000000e+00 : f64,
              formats = ["Dense", "Dense", "Dense"],
              indexing_maps = [#map0, #map1, #map2]} :
              (tensor<296x312x312xf64>,
              tensor<312x296x312xf64>,
              tensor<312x312xf64>) -> ()
```

$$C_{ba} = \sum_{cd} A_{cad} \cdot B_{dcb}$$

```
%3 = alloc() {alignment = 32 : i64} : memref<312x312x296xf64>
linalg.copy(%0, %3) {inputPermutation = #map0,
                     outputPermutation = #map1} :
                     memref<296x312x312xf64>, memref<312x312x296xf64>
```

$A_{cad} \Rightarrow A_{a(cd)}$

```
%4 = alloc() {alignment = 32 : i64} : memref<312x312xf64>
linalg.copy(%2, %4) {inputPermutation = #map2,
                     outputPermutation = #map3} :
                     memref<312x312xf64>, memref<312x312xf64>
```

$B_{dcb} \Rightarrow B_{(cd)b}$

```
%5 = linalg.reshape %3 [#map4, #map5] : memref<312x312x296xf64> into memref<312x92352xf64>
%6 = linalg.reshape %1 [#map6, #map7] : memref<312x296x312xf64> into memref<92352x312xf64>

linalg.matmul  {__alpha__ = 1.000000e+00 : f64,
                __beta__ = 0.000000e+00 : f64,
                __internal_linalg_transform__ = "__with_tiling__"} %5, %6, %4 :
                (memref<312x92352xf64>, memref<92352x312xf64>, memref<312x312xf64>)
```

GEMM

```
linalg.copy(%4, %2) {inputPermutation = #map3,
                     outputPermutation = #map2} :
                     memref<312x312xf64>, memref<312x312xf64>
```

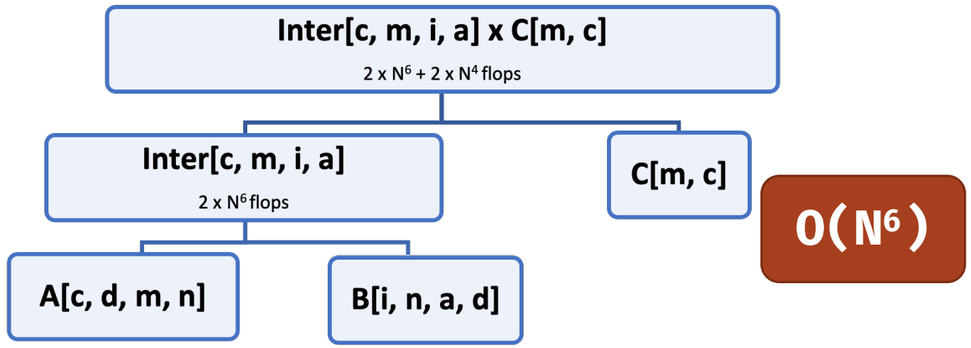
$C_{ab} \Rightarrow C_{ba}$

High Level Optimizations

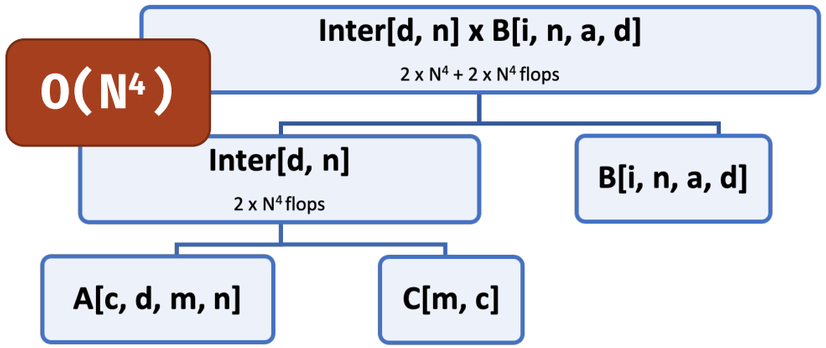
- CCSD-T1 consists of chain of tensor contractions (up to 5 tensors)
- Multi-operand operation count optimization
  - Different ordering of tensor contractions can have performance consequences
  - Optimal ordering reduces the number of operations to be performed
  - Consider the cost of transpose (outcome of TTGT)

```
D[a,i]= A[c,d,m,n] x B[i,n,a,d] x C[m,c]
```

```
D[a,i]= (A[c,d,m,n] x B[i,n,a,d]) x C[m,c]
```




```
D[a,i]= (A[c,d,m,n] x C[m,c]) x B[i,n,a,d]
```

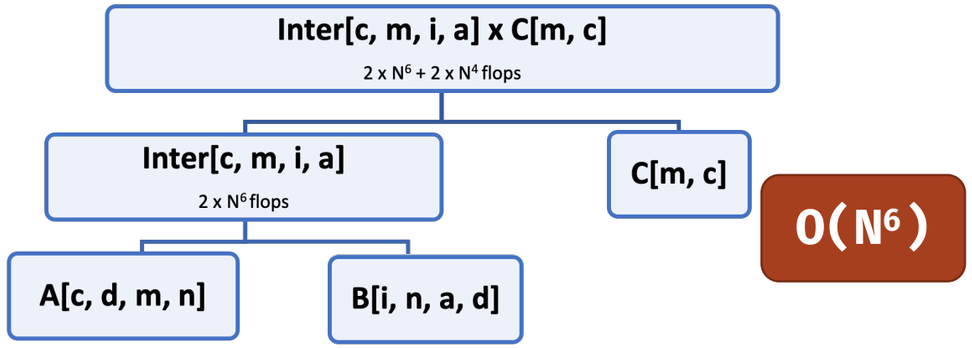


High Level Optimizations

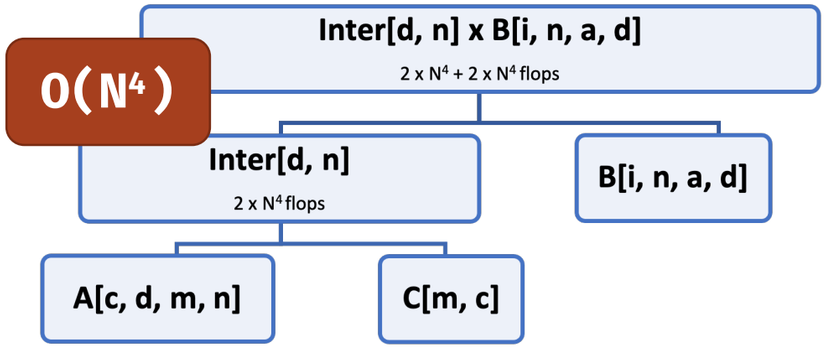
- Multi-operand operation count optimization
  - Different ordering can have performance consequences

```
D[a,i]= A[c,d,m,n] x B[i,n,a,d] x C[m,c]
```

```
D[a,i]= (A[c,d,m,n] x B[i,n,a,d]) x C[m,c]
```




```
D[a,i]= (A[c,d,m,n] x C[m,c]) x B[i,n,a,d]
```



- Best permutation for minimizing **transpose** cost
  - Compute GEMM indices M-N-K indices for the given tensor contraction
  - For each combination of M, N, and K indices including swapping A and B tensors
    - ✓ Compute the transpose cost
  - Find best candidate by comparing the transpose time

Low Level Optimizations

- Optimizing **Transpose** operation
  - N-dimensional tensor transpose has N! distinct ways to order the loops
  - Optimal loop ordering leads significant performance difference
  - A heuristic[1] that reorders loops to put innermost indices close

- Optimizing **GEMM** operation
  - We follow the BLIS[2]-like tiling optimizations
    - ✓ LHS matrix (%A) of size M_C x K_C reused in L3 cache,
    - ✓ RHS matrix (%B) tile of size K_C x N_R reused L2 cache,
    - ✓ Output matrix (%C) M_R x N_R reused just the registers.
  - Call BLIS-micro kernel after cache tiling

[1] HPTT: a high-performance tensor transposition C++ library. Paul Springer, Tong Su, Paolo Bientinesi. 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, June 2017
[2] The BLIS Framework: Experiments in Portability. F. G. Van Zee et. al. ACM Transactions on Mathematical Software

CCSD-T1 method in COMET DSL

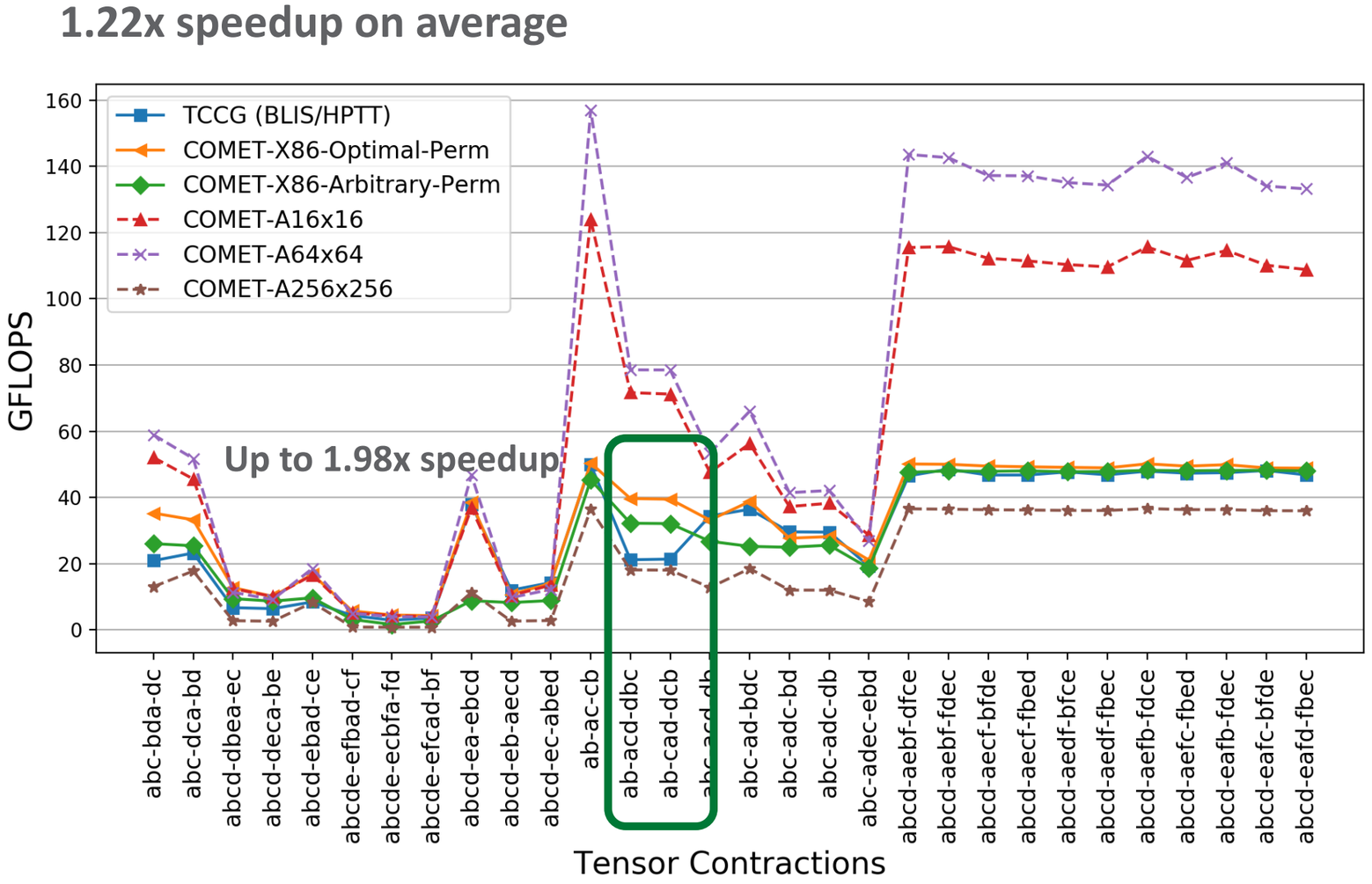
```
4   def ccsd_t1() {
5   # IndexLabel Declarations
6     IndexLabel [i, j, m, n] = [0:72];
7     IndexLabel [a, b, c, d] = [0:72];
8
9   # Tensor Declarations
10    Tensor<double> i0[i, a];
11    Tensor<double> f[i, a];
12    Tensor<double> t1[i, a];
13    Tensor<double> t2[i, j, a, b];
14    Tensor<double> v[a, b, i, j];
15
16  # Tensor Operations
17    i0[i, a] -= f[i, a];                                    # 1
18    i0[i, a] += f[i, m] * t1[m, a];                         # 2
19    i0[i, a] -= f[c, a] * t1[i, c];                         # 3
20    i0[i, a] -= 2 * v[c, i, m, a] * t1[m, c];               # 4
21    i0[i, a] += v[i, c, m, a] * t1[m, c];                   # 5
22    i0[i, a] -= 2 * f[m, c] * t2[m, i, c, a];               # 6
23    i0[i, a] += f[m, c]* t2[i, m, c, a];                    # 7
24    i0[i, a] += f[m, c] * t1[i, c] * t1[m, a];              # 8
25    i0[i, a] += 2 * v[c, d, m, n] * t2[m, n, a, d] * t1[i, c];      # 9
26    i0[i, a] += 2 * v[c, d, m, n] * t2[i, n, c, d] * t1[m, a];      # 10
27    i0[i, a] -= 4 * v[c, d, m, n] * t2[i, n, a, d] * t1[m, c];      # 11
28    i0[i, a] += 2 * v[c, d, m, n] * t2[i, n, d, a] * t1[m, c];      # 12
29    i0[i, a] += 2 * v[c, d, m, n] * t1[i, d] * t1[n, a]* t1[m, c];  # 13
30    i0[i, a] -= v[d, c, m, n] * t2[m, n, a, d] * t1[i, c];          # 14
31    i0[i, a] -= v[d, c, m, n] * t2[i, n, c, d] * t1[m, a];          # 15
32    i0[i, a] += 2 * v[d, c, m, n] * t2[i, n, a, d] * t1[m, c];      # 16
33    i0[i, a] -= v[d, c, m, n] * t2[i, n, d, a] * t1[m, c];          # 17
34    i0[i, a] -= v[d, c, m, n] * t1[i, d] * t1[n, a] * t1[m, c];     # 18
35    i0[i, a] += 2 * v[c, i, m, n] * t2[m, n, c, a];                 # 19
36    i0[i, a] += 2 * v[c, i, m, n] * t1[m, c] * t1[n, a];            # 20
37    i0[i, a] -= v[i, c, m, n] * t2[m, n, c, a];                     # 21
38    i0[i, a] -= v[i, c, m, n] * t1[m, c] * t1[n, a];                # 22
39    i0[i, a] -= 2 * v[c, d, m, a] * t2[m, i, c, d];                 # 23
40    i0[i, a] -= 2 * v[c, d, m, a] * t1[m, c] * t1[i, d];            # 24
41    i0[i, a] += v[d, c, m, a] * t2[m, i, c, d];                     # 25
42    i0[i, a] += v[d, c, m, a] * t1[m, c] * t1[i, d];                # 26
43
44    return i0;
45  }
46
47  def main() {
48    var i0 = ccsd_t1();
49    print(i0);
50  }
```

Coupled-Cluster (CC) Theory

- CCSD is the most common CC method and the principle method that higher-order approximation and excited state methods are built upon.
- CCSD involves iteratively solving a set of non-linear equations.
- Computational complexity: N^6, Memory complexity: N^4

Dense Tensor Contractions Evaluation – Comparison with Hand Optimized

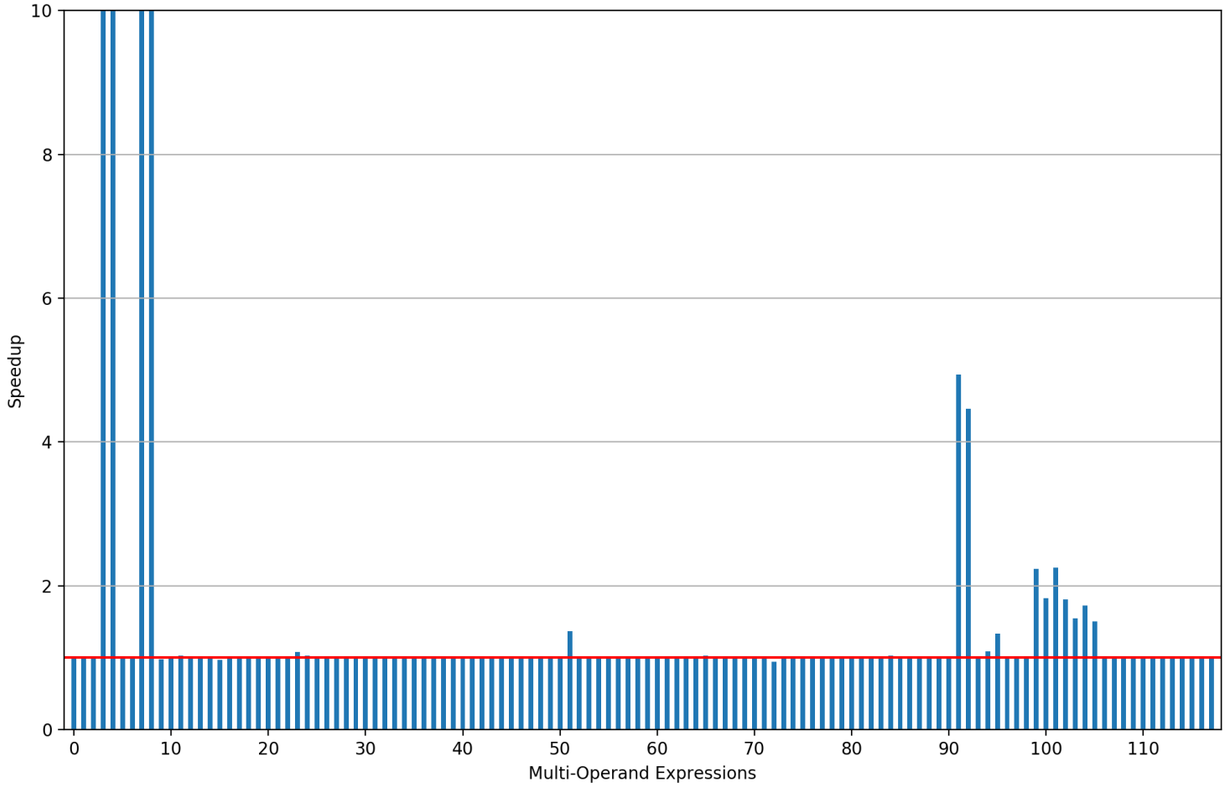
- State-of-the-art libraries
  - TCCG[1] bechmarks
  - BLIS[2] (for GEMM operation)
  - HPTT[3] (for Transpose operation)
- Performance
  - Improve in some cases by **1.98x speedup**
  - On par with hand tuned implementations - **1.22x speedup** on **average**



[1] P. Springer, and P. Bientinesi. “Design of a high-performance GEMM-like Tensor-Tensor Multiplication”. *CoRR*, 2016.
[2] F. G. Van Zee et. al. “The BLIS Framework: Experiments in Portability” ACM Transactions on Mathematical Software 2016
[3] P. Springer, T. Su, and P. Bientinesi. “HPTT: A High-Performance Tensor Transposition C++ Library” ARRAY 2017

Multi-operand Optimization Performance

We evaluated 118 dense tensor contractions from NWChem[1] methods (CCSD-T1 and CCSD-T2) that involve 3 and 4 operands



Multi-operand Tensor expressions	Perf.
A[c,d,m,n] * B[i,n,a,d] * C[m,c]	23.9
A[d,c,m,n] * B[i,n,a,d] * C[m,c]	21.1
A[c,d,m,n] * B[i,d] * C[n,a] * D[m,c]	4.9
A[d,c,m,n] * B[i,d] * C[n,a] * D[m,c]	4.5
A[m,n,e,j] * B[e,i] * C[a,m] * D[b,n]	1.4
A[m,n,f,e] * B[e,i] * C[f,n] * D[a,b,m,j]	1.4
A[m,n,e,f] * B[a,m] * C[f,n] * D[e,b,i,j]	2.2
A[m,n,e,f] * B[b,m] * C[f,n] * D[e,a,j,i]	1.8
A[n,m,e,f] * B[a,m] * C[f,n] * D[e,b,i,j]	2.2
A[n,m,e,f] * B[b,m] * C[f,n] * D[e,a,j,i]	1.8
A[m,n,e,f] * B[e,i] * C[f,n] * D[a,b,m,j]	1.5
A[m,n,e,f] * B[e,j] * C[f,n] * D[b,a,m,i]	1.7
A[m,n,f,e] * B[e,j] * C[f,n] * D[b,a,m,i]	1.5

1- E. Apra et. al. “NWChem: Past, present, and future”, *The Journal of Chemical Physics* **152**, 184102 (2020).

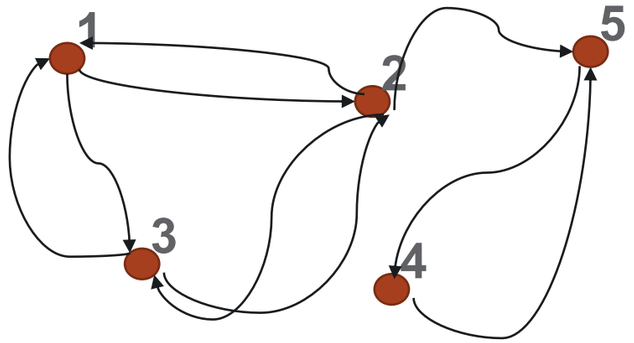
Pacific Northwest
NATIONAL LABORATORY

COMET Sparse Tensor Algebra

Graph Algorithms and Linear Algebra

- A graph as a matrix
  - Adjacency Matrix - A square sparse matrix where rows and columns are labeled by vertices and non-zero elements present edges from one vertex to another one

	1.	2.	3.	4.	5.
1.		1	1		
2.	1		1		1
3.	1	1			
4.					1
5.				1	



- Linear algebra (LA) provide an elegant, concise, intuitive, and portable programming abstraction for designing graph algorithms
- 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

Pacific Northwest
NATIONAL LABORATORY

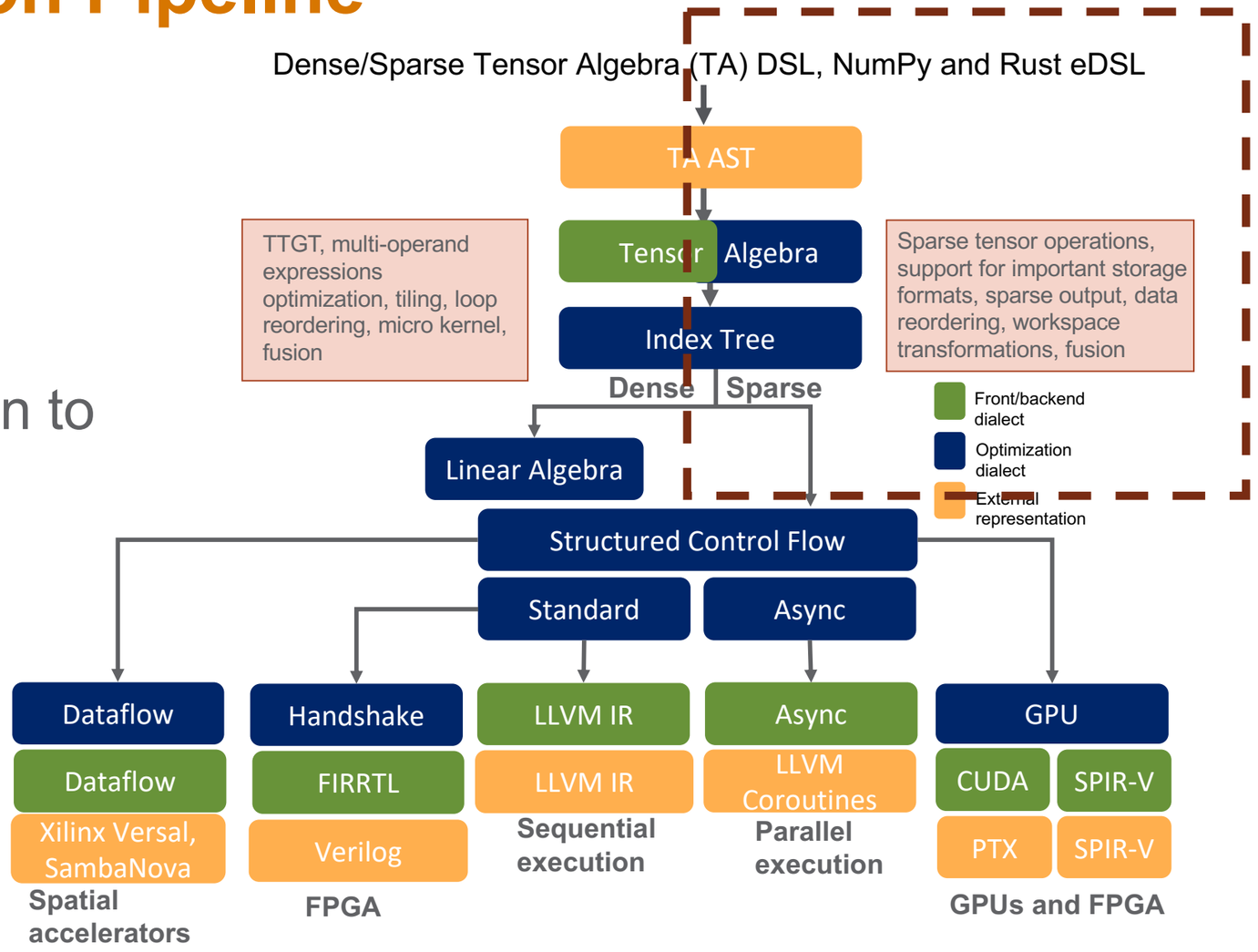
Sparse Computations

- 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]

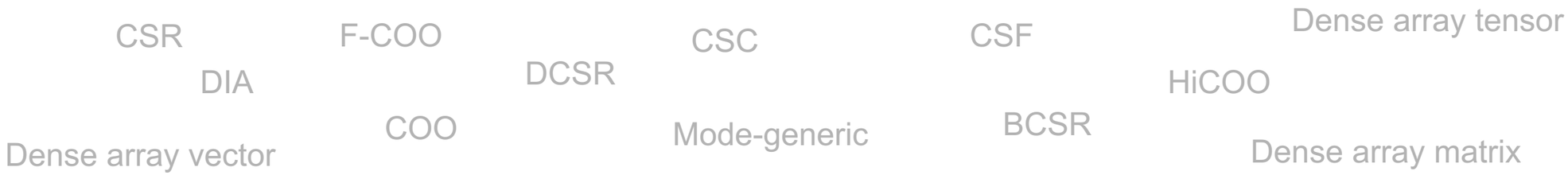
- Internal sparse tensor storage format
- Sparse data type
- An attribute per tensor dimension to support sparse tensor storage format
- Automatic code generation for sparse tensor operations
- Support for sparse output
- Input-dependent optimization
  - Data reordering to enhance data locality



[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.

Internal Tensor Storage (Dense/Sparse)

- There are many different sparse formats to store matrices/tensors



- The computation kernels based on different storage formats present ***different code structure and performance***
- We define the following four storage ***format attributes***[1] to represent ***each tensor dimension***:
  - **Dense (D):** all the coordinates in this dimension are accessed during the computations.
  - **Compressed-Unique (C):** Nonzero coordinates in this dimension are compressed and only the unique coordinated are listed.
  - **Compressed-Nonunique (CN):** All the nonzero coordinates are stored in one array, and the start and end position of the coordinates stored in another auxiliary array.
  - **Singleton (S):** All the nonzero coordinates are listed without any other information.
- Currently, we support common matrix and tensor formats in the COMET compiler: ***COO, CSR, DCSR, BCSR, ELLPACK,CSF, Mode-generic.***

[1] Format Abstraction for Sparse Tensor Algebra Compilers. Stephen Chou and Fredrik Kjolstad, and Saman Amarasinghe, OOPSLA, November, 2018.

Example Internal tensor storage

Sparse Matrix A

	0	1	2	3
0	1			2
1	3	4		
2				
3		5		
4			6	7

row (D) { row_pos: 5 }

column (C) { col_pos: 0 2 4 4 5 7 ; col_crd: 0 3 0 1 1 2 3 }

A val: 1 2 3 4 5 6 7

CSR: A[Dense, Compressed-Unique]

row (CN) { row_pos: 0 7 ; row_crd: 0 0 1 1 3 4 4 }

column (S) { col_crd: 0 3 0 1 1 2 3 }

A val: 1 2 3 4 5 6 7

COO: A[Compressed-Nonunique, Singleton]

row (C) { row_pos: 0 4 ; row_crd: 0 1 3 4 }

column (C) { col_pos: 0 2 4 5 7 ; col_crd: 0 3 0 1 1 2 3 }

A val: 1 2 3 4 5 6 7

DCSR: A[Comp.-Unique Comp.-Unique]

Sparse Code Generation Example

Pacific Northwest NATIONAL LABORATORY

Sparse Matrix dense Matrix multiplication: $Cij = \sum_{k} Aik * Bkj$

	0	1	2	3
0	1			2
1	3	4		
2				
3		5		
4			6	7

row_pos	0	7					
row_crd	0	0	1	1	3	4	4
col_crd	0	3	0	1	1	2	3
A val	1	2	3	4	5	6	7

COO: A[CN, S]

row_pos	5						
col_pos	0	2	4	4	5	7	
col_crd	0	3	0	1	1	2	3
A val	1	2	3	4	5	6	7

CSR: A[D, CU]

row_pos	0	4					
row_crd	0	1	3	4			
col_pos	0	2	4	5	7		
col_crd	0	3	0	1	1	2	3
A val	1	2	3	4	5	6	7

DCSR: A[CU, CU]

```
for(int m = row_pos[0]; m < row_pos[1]; m++){
    int i = row_crd[m];
    int j = col_crd[m];
    for(int k=0; k < 32; k++){
        C[i][k] += Aval[m] * B[j][k];
    }
}
```

```
for(i = 0; i < row_pos[0]; i++){
    for(n = col_pos[i]; n < col_pos[i+1]; n++){
        int j = col_crd[n];
        for(int k=0; k < 32; k++){
            C[i][k] += Aval[n] * B[j][k];
        }
    }
}
```

```
for(m = row_pos[0]; m < row_pos[1]; m++){
    int i = row_crd[m];
    for(n = col_pos[i]; n < col_pos[i+1]; n++){
                int j = col_crd[n];
                for(int k=0; k < 32; k++){
            C[i][k] += Aval[n] * B[j][k];
        }
    }
}
```

Test Case: SpMM

```
"ta.tc"(%3, %4, %5) {__alpha__ = 1.000000e+00 : f64,
                     __beta__ = 0.000000e+00 : f64,
                     formats = ["COO", "Dense", "Dense"],
                     indexing_maps = [#map0, #map1, #map2]} :
                     (tensor<?x?xf64>, tensor<?x32xf64>,
                     tensor<?x32xf64>) -> ()
```

COO code generation

```
scf.for %arg0 = %30 to %32 step %c1 {
  %34 = load %18[%arg0] : memref<?xi32>
  %35 = index_cast %34 : i32 to index
  %36 = load %22[%arg0] : memref<?xi32>
  %37 = index_cast %36 : i32 to index
  scf.for %arg1 = %c0 to %c32 step %c1 {
    %38 = load %24[%37] : memref<?xf64>
    %39 = load %26[%37, %arg1] : memref<?x32xf64>
    %40 = load %27[%35, %arg1] : memref<?x32xf64>
    %41 = mulf %38, %39 : f64
    %42 = addf %40, %41 : f64
    store %42, %27[%35, %arg1] : memref<?x32xf64>
  }
}
```

```
"ta.tc"(%3, %4, %5) {__alpha__ = 1.000000e+00 : f64,
                     __beta__ = 0.000000e+00 : f64,
                     formats = ["CSR", "Dense", "Dense"],
                     indexing_maps = [#map0, #map1, #map2]} :
                     (tensor<?x?xf64>, tensor<?x32xf64>,
                     tensor<?x32xf64>) -> ()
```

CSR code generation

```
scf.for %arg0 = %c0 to %30 step %c1 {
  %32 = addi %arg0, %c1 : index
  %33 = load %20[%arg0] : memref<?xi32>
  %34 = index_cast %33 : i32 to index
  %35 = load %20[%32] : memref<?xi32>
  %36 = index_cast %35 : i32 to index
  scf.for %arg1 = %34 to %36 step %c1 {
    %37 = load %22[%arg1] : memref<?xi32>
    %38 = index_cast %37 : i32 to index
    scf.for %arg2 = %c0 to %c32 step %c1 {
      %39 = load %24[%38] : memref<?xf64>
      %40 = load %26[%38, %arg2] : memref<?x32xf64>
      %41 = load %27[%arg0, %arg2] : memref<?x32xf64>
      %42 = mulf %39, %40 : f64
      %43 = addf %41, %42 : f64
      store %43, %27[%arg0, %arg2] : memref<?x32xf64>
    }
  }
}
```

Pacific Northwest NATIONAL LABORATORY

Sparse Computation

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

Challenges

- Sparse tensors are stored in compressed irregular data structure, which introduces irre
- Sto
- Co other operations
- Sparse output contains expensive insertions and accesses to sparse tensors, which has large time complexity

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

Storing outputs in sparse format is complicated but it is necessary to avoid ***densification***

Support for Sparse Outputs

- 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**

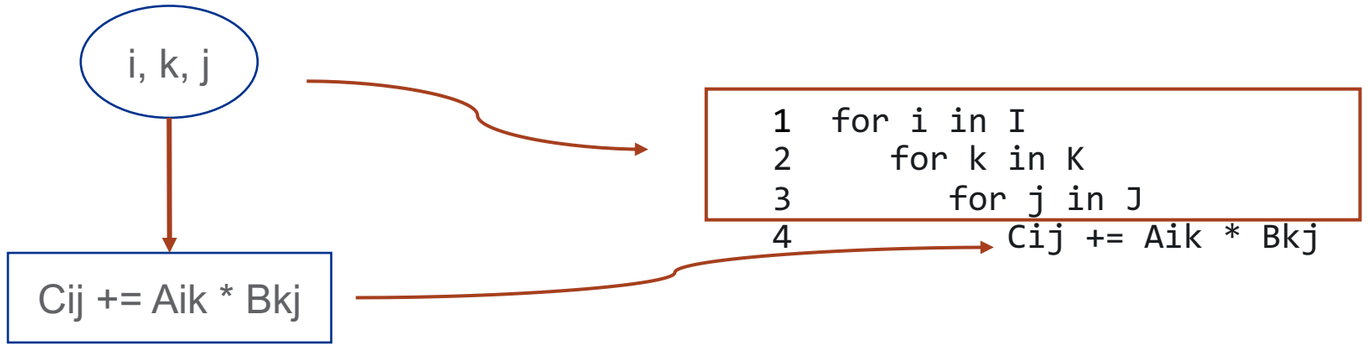
- This approach brings the following advantages:
  - Significantly improves performance of sparse kernels through efficient dense data structures accesses.
  - Reduces memory footprint
  - Avoids “densifying” issue in the compound expressions

[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

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

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

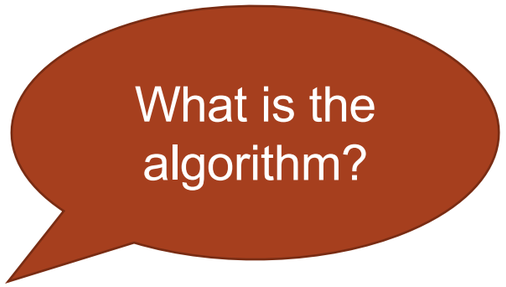


Index tree for SpGEMM

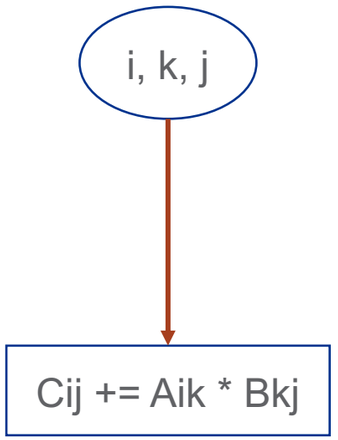
Pseudo-code for SpGEMM

Workspace Transformation

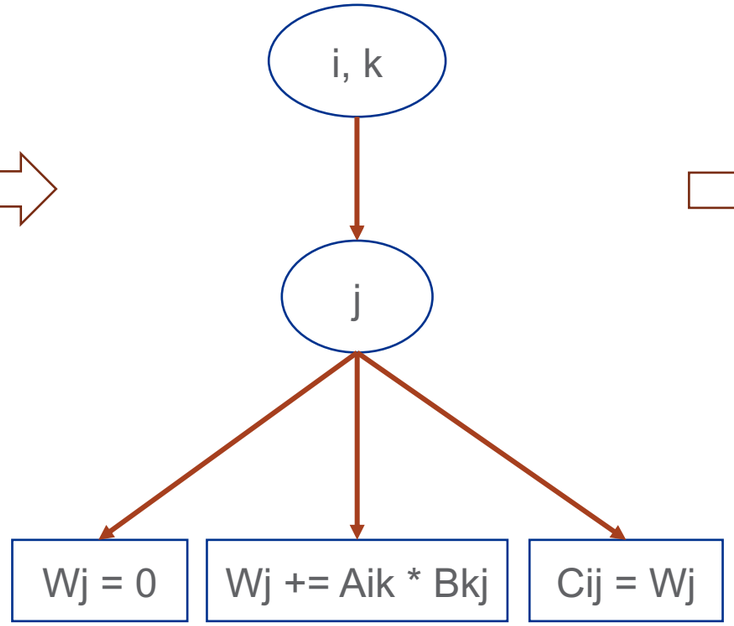
- We perform compiler transformation in the index tree representation of a tensor expression
  - Benefits
    - ✓ Reduces expensive insertions/ accesses to sparse tensors
      - Dense data structure has better locality
      - Generates “for” loops instead of “while” loops
      - Utilize the existing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



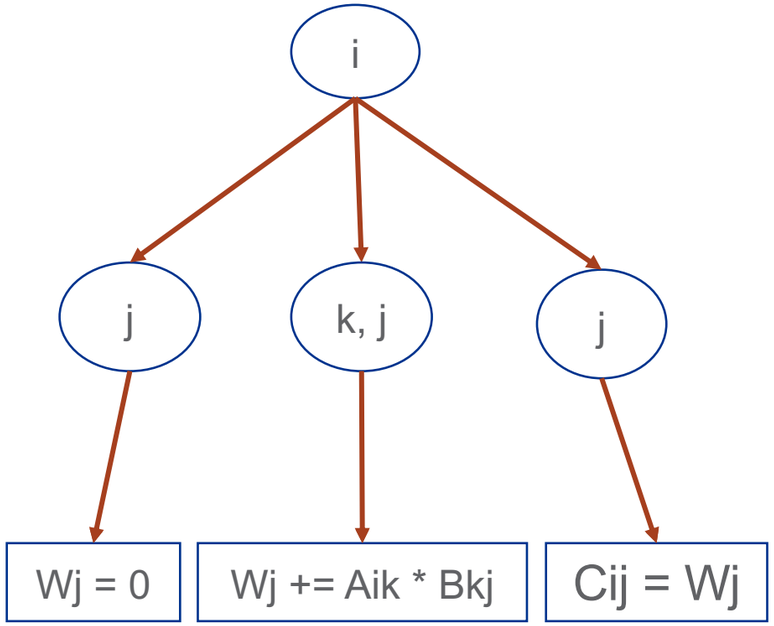
Transformation in Index Tree



Index tree for SpGEMM

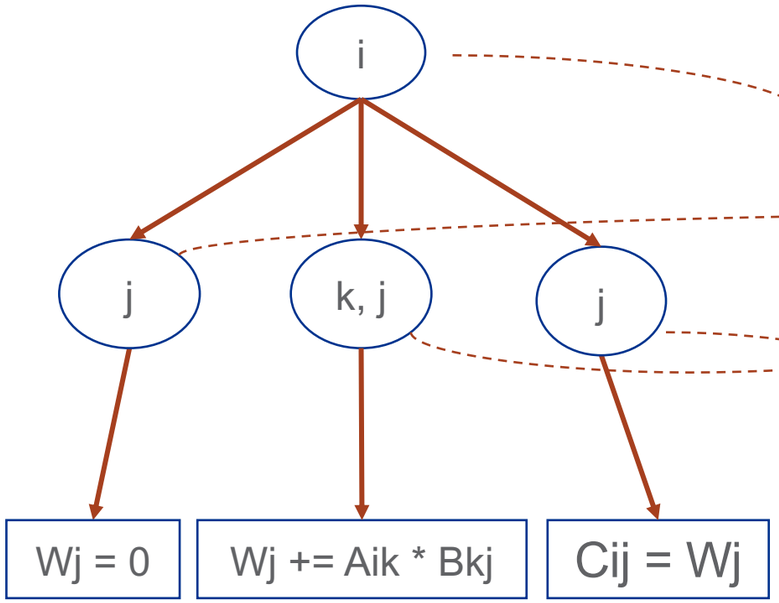


Index tree for SpGEMM with workspace



Eliminate loop invariant redundancy

Code Generation from Index Tree IR Operations



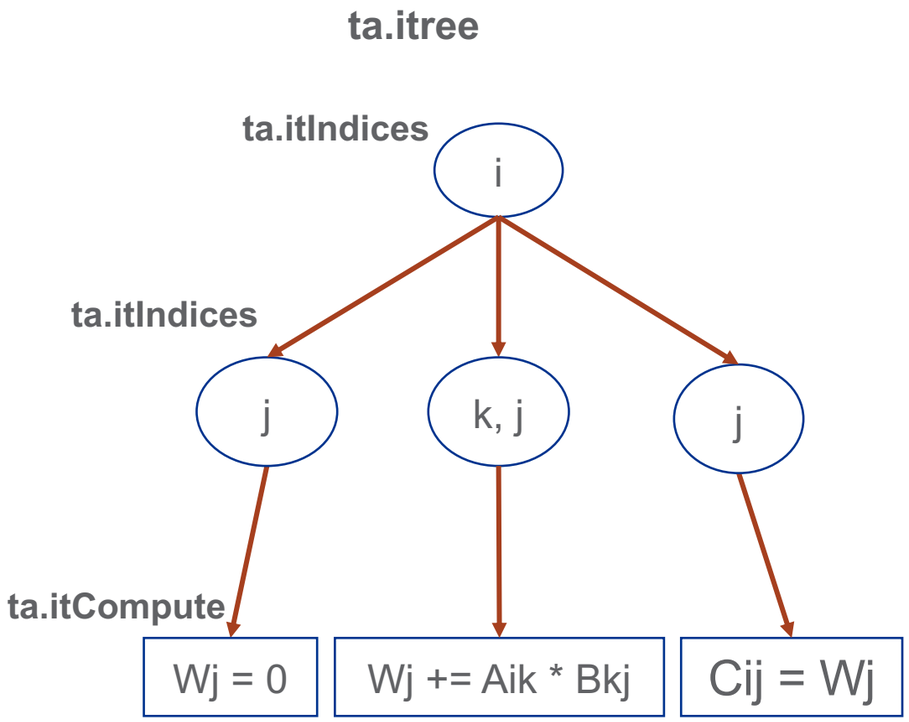
Index tree for SpGEMM

```
1  for i in I
2     for j in J
3        Wj = 0
4     for k in K
5        for j in J
6           Wj += Aik * Bkj
7     for j in J
8        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

```
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
```

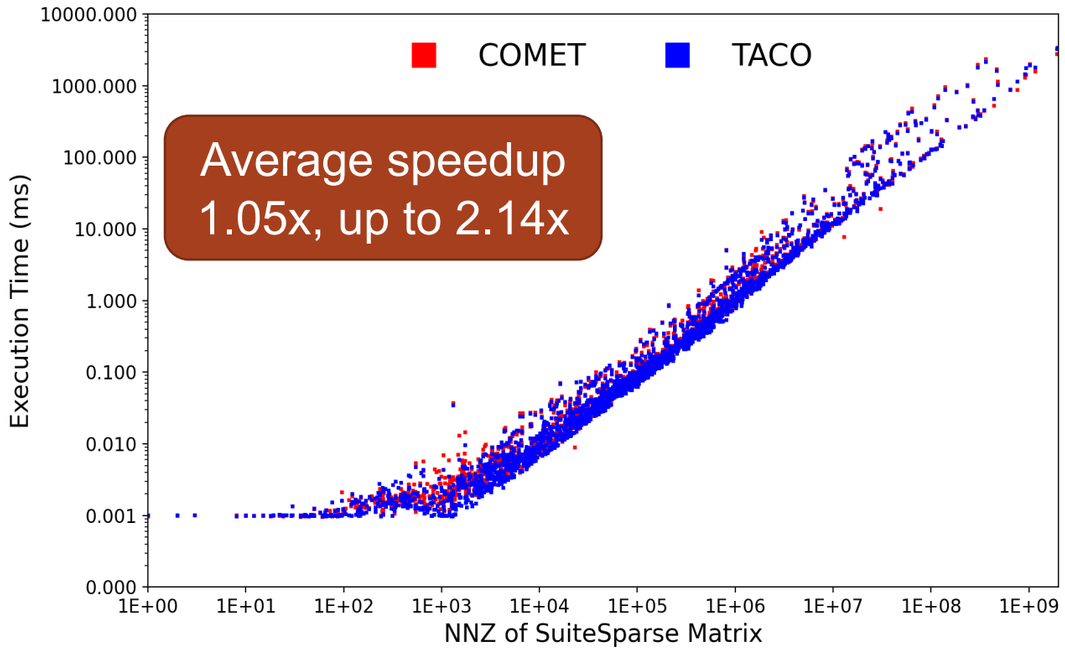
SpGEMM Index Tree IR Ops

```
1  for i in I
2     for j in J
3        Wj = 0
4     for k in K
5        for j in J
6           Wj += Aik * Bkj
7     for j in J
8        Cij = Wj
      // update pos/crd
```

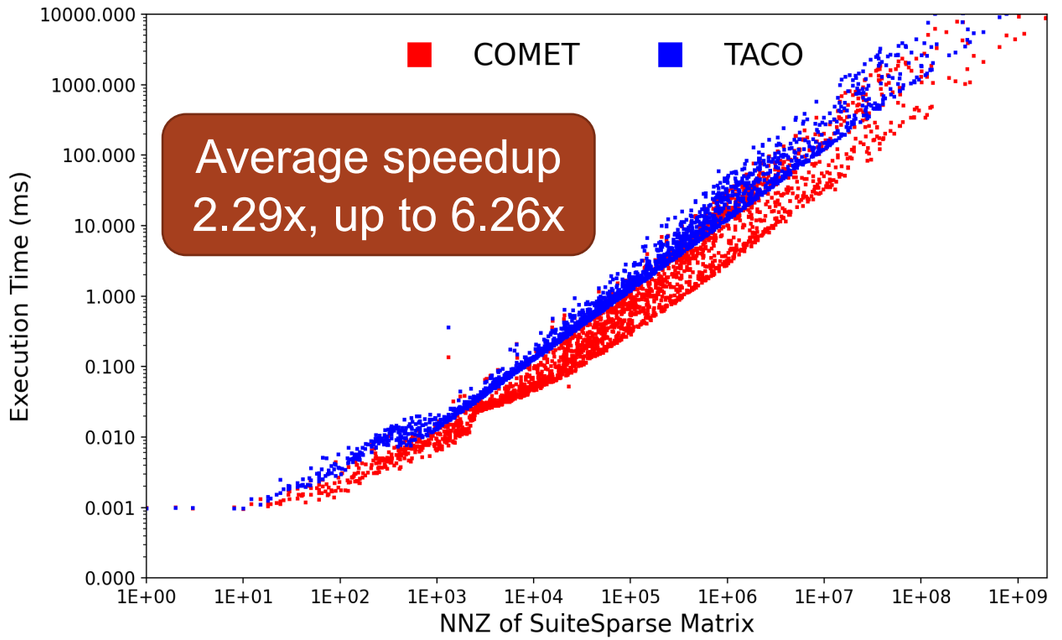
Pseudo-code for SpGEMM with workspace

Mixed mode Sparse Kernels

- TACO[1] is a library that automatically generates efficient code for sparse and dense tensor algebra computations.
- We compare the performance of automatically generated COMET sparse kernels with the TACO compiler by using 2840 matrices from Suite Sparse Matrix Collection dataset[2]
- COMET significantly outperforms TACO due to a better vectorization and loop unrolling



Sparse Matrix Dense Vector Multiplication (SpMV)

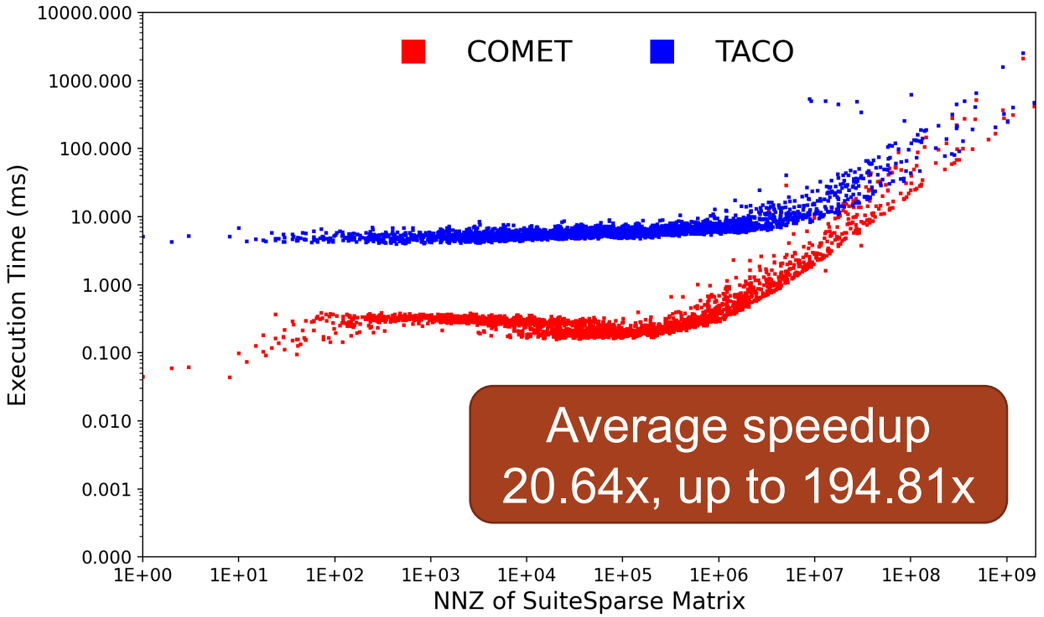


Sparse Matrix Dense Matrix Multiplication (SpMM)

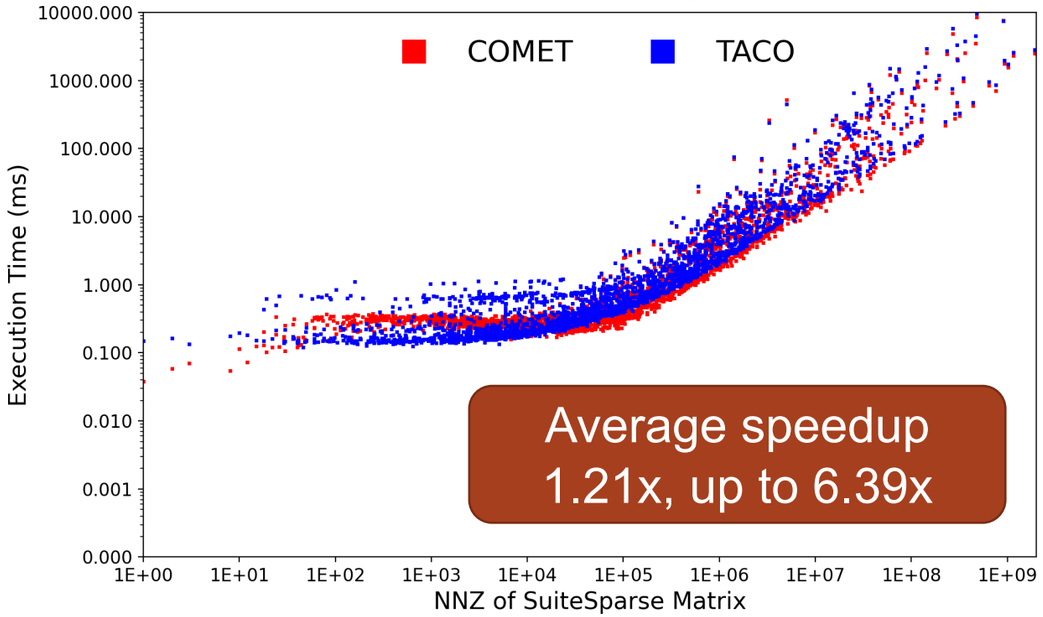
[1] TACO: The Tensor Algebra Compiler - http://tensor-compiler.org
[2] SuiteSparse Matrix Collection - https://sparse.tamu.edu

Mixed mode Sparse Kernels (Parallel execution)

- We use asynchronous/multi-threaded programs in a sequential manner and use LLVM compiler passes to convert asynchronous functions into LLVM coroutines (Async Dialect).
- For parallel SpMV, COMET achieves an average of 20.64x speedup over TACO with 24 parallel threads
  - Performance difference is due to the overhead introduced by the underlying parallel runtime
  - COMET uses an asynchronous task-based programming model while TACO leverages OpenMP



Sparse Matrix Dense Vector Multiplication (SpMV)

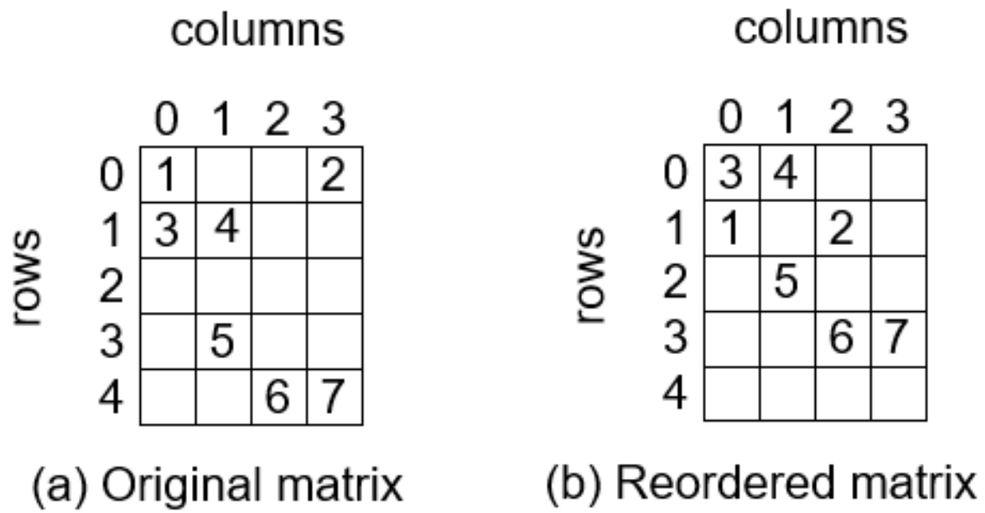


Sparse Matrix Dense Matrix Multiplication (SpMM)

1: TACO: The Tensor Algebra Compiler - http://tensor-compiler.org
2: SuiteSparse Matrix Collection - https://sparse.tamu.edu

Input dependent Optimization: Data Reordering

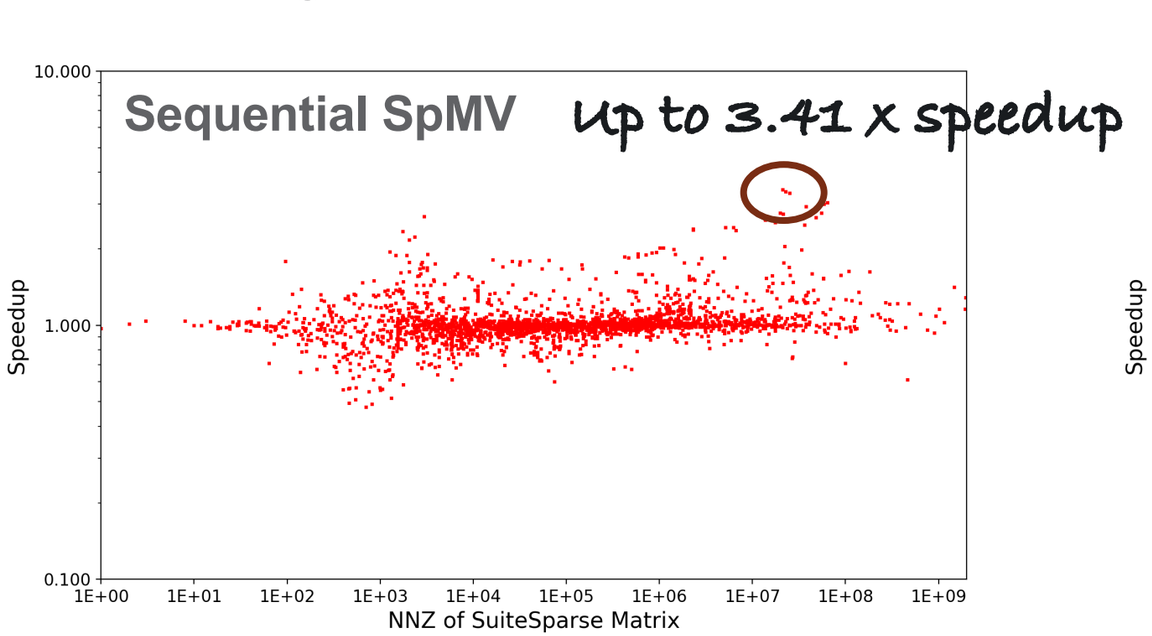
- We integrate sparse tensor reordering into the compiler
  - relabels the matrix indices to reorganize the nonzero structure of the matrix
  - target improving the spatial and temporal locality of sparse operations
- We use the state-of-the-art sparse tensor reordering algorithm -- Lexi-Order[1]:
  - sorts a specific dimension (either rows or columns for matrices) in an iteration
  - uses the doubly lexical ordering algorithm, and sorts all dimensions in turn across iteration

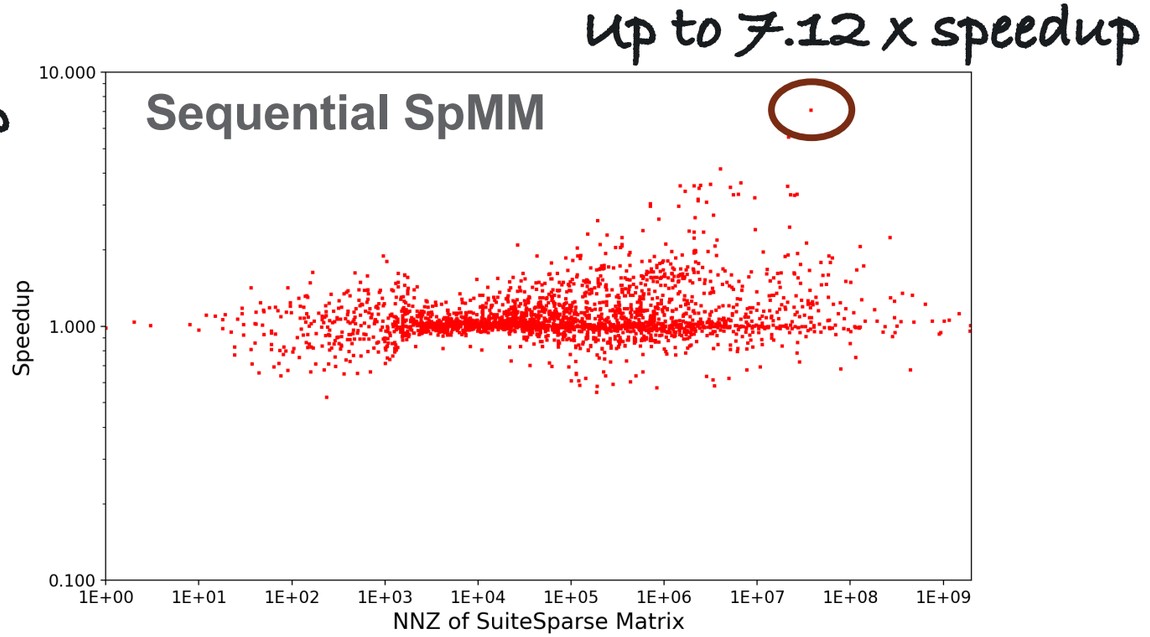


1- Efficient and Effective Sparse Tensor Reordering. *Jiajia Li*, Bora Ucar, Umit V. Catalyurek, Jimeng Sun, Kevin Barker, Richard Vuduc. International Conference on Supercomputing (ICS). 2019

Data reordering Performance

- COMET attempts to increase spatial and temporal locality to achieve higher performance by reordering data[1] in memory.
- We compare COMET performance *with* and *without* data reordering for sequential SpMV and SpMM with various input matrices
- Our results show that, in many cases, there is significant performance advantage of data reordering.

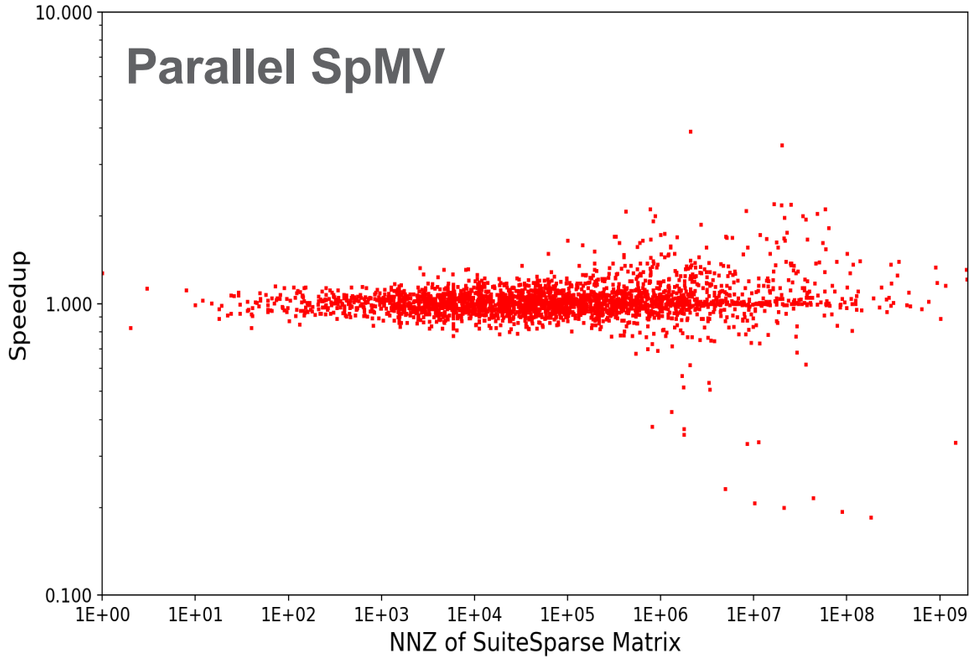


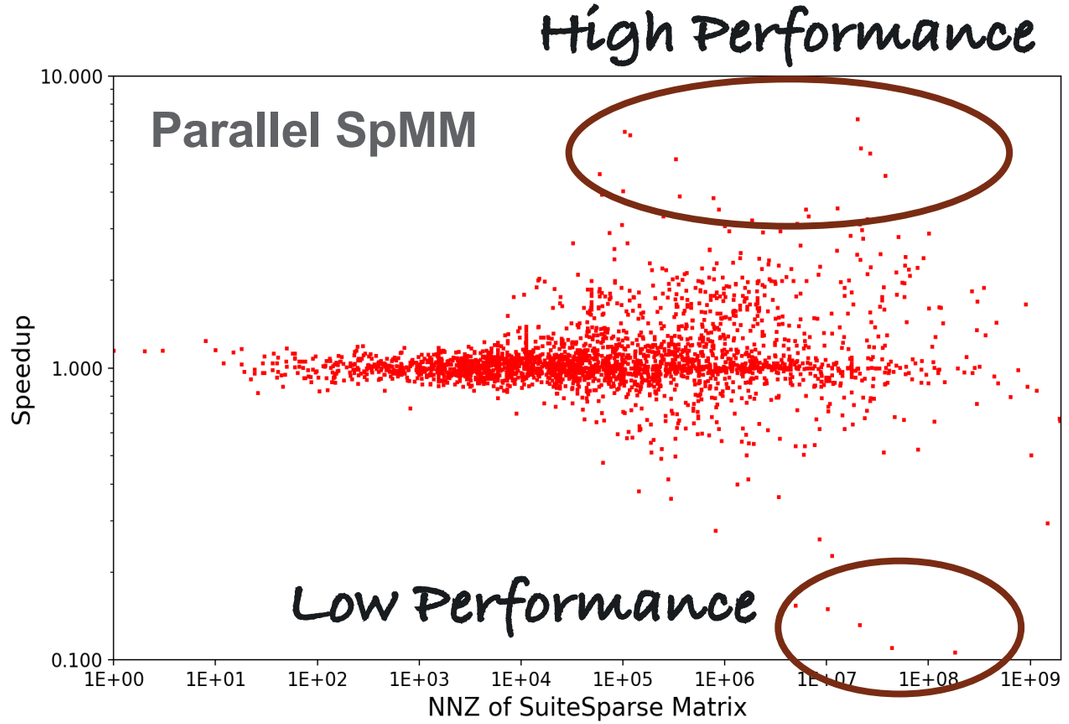


[1] J. Li, B. Uc ̧ar, ̈U. V. Catalyurek, J. Sun, K. Barker, and R. Vuduc, “Efficient and effective sparse tensor reordering,” in Proceedings of the ACM International Conference on Supercomputing, 2019

Data reordering Performance (Parallel)

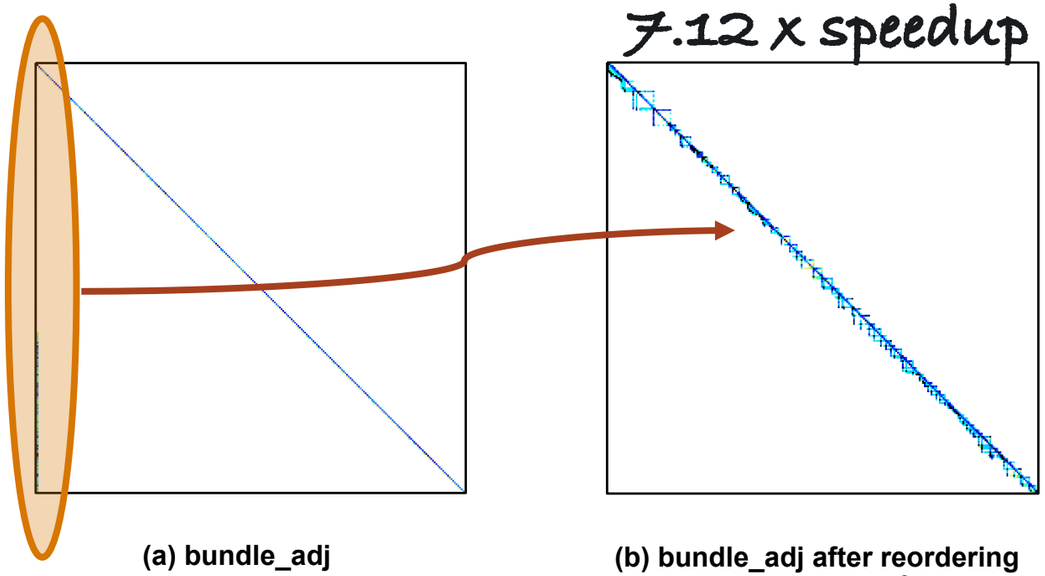
- COMET achieves significant performance improvement for parallel execution, up to 3.89x (average1.03x), and 7.14x (average 1.13x) for SpMV and SpMM, respectively.
- We also observe that there might be some performance degradation





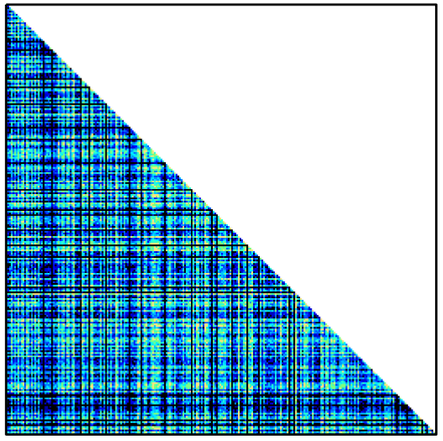
Matrices with and without reordering

- Performance improvement or degradation depend on the input
- We analyzed how the original sparsity patterns affect reordering and performance
- **High-Performance (bundle_adj):**
  - the nonzero elements are clustered around the diagonal.
  - Similar work distribution among threads
- **Low-Performance (kroon_g500-logn20):**
  - the nonzeros are clustered around the top-left corner
  - Load-imbalance among parallel threads.

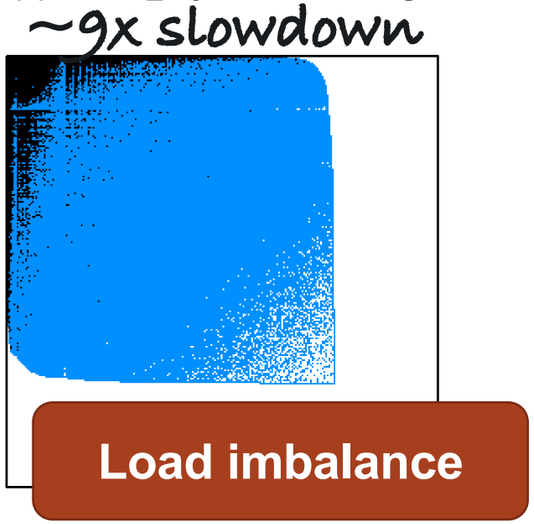


(a) bundle_adj

(b) bundle_adj after reordering

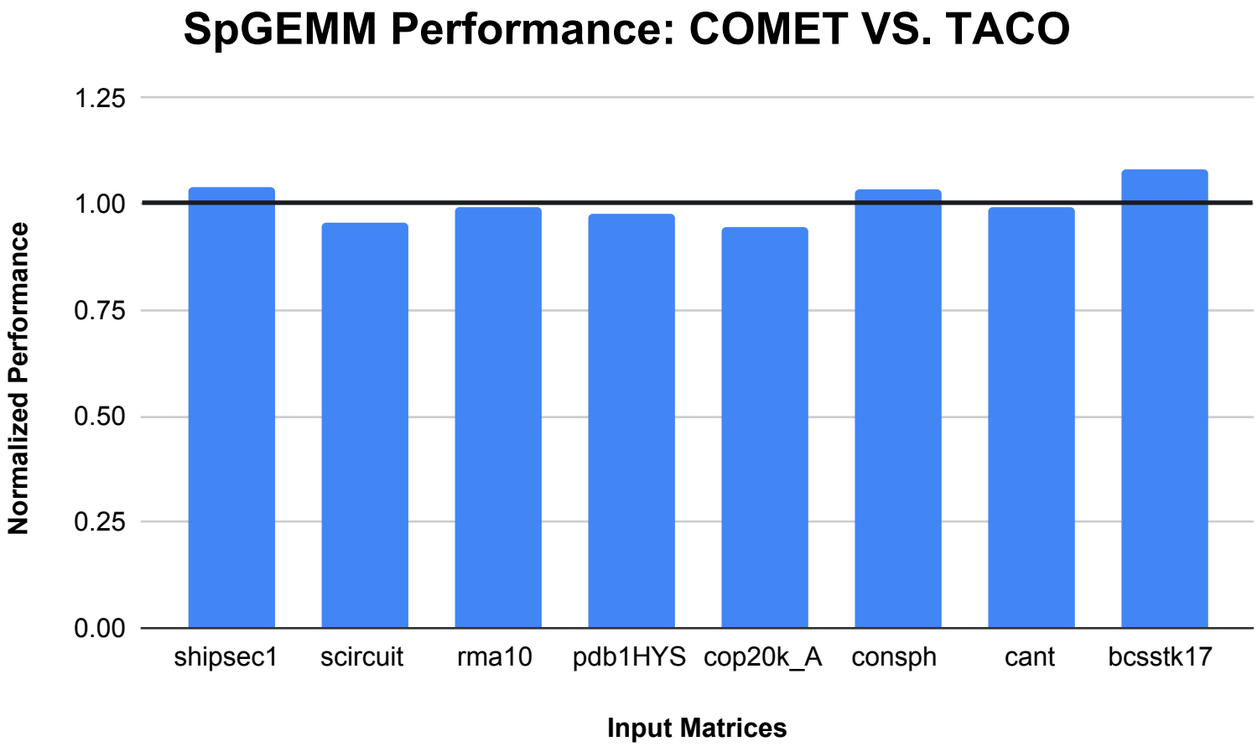


(c) kron_g500-logn20



(d) kron_g500-logn20 after reordering

Sparse-Sparse Kernel (e.g., SpGEMM)



- SpGEMM, sparse matrix – sparse matrix multiplication
- COMET's performance is comparable to the state-of-the-art solutions
- Performance difference in the range of variation

Pacific Northwest
NATIONAL LABORATORY

COMET Hands-on Session

Pacific Northwest NATIONAL LABORATORY

Testcase 1: Intensli1

```
def main() {
    #IndexLabel Declarations
    IndexLabel [a] = [312];
    IndexLabel [c] = [296];
    IndexLabel [b] = [312];
    IndexLabel [d] = [312];

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

    #Tensor Data Initialization
    A[d, c, a] = 2.2;
    B[b, d] = 3.4;
    C[a, b, c] = 0.0;

    #Tensor Contraction
    C[a, b, c] = A[d, c, a] * B[b, d];
}
```

Lowering a tensor contraction in a native **loop form**

```
$BUILD_DIR/bin/comet  \
    -lower-ta-to-loops \
    $BENCH_DIR/intensli1.ta \
    &>  $BENCH_IRs/intensli1_case1_loop.mlir
```

Execution time: 7.002250 TIME

Testcase 3: CCSD_T1_11

```
def main() {
    #IndexLabel Declarations
    IndexLabel [i, j, m, n] = [0:72];
    IndexLabel [a, b, c, d] = [0:72];

    #Tensor Declarations
    Tensor<double> i0([i, a], {Dense});
    Tensor<double> t1([i, a], {Dense});
    Tensor<double> t2([i, j, a, b], {Dense});
    Tensor<double> v([a, b, i, j], {Dense});

    #Tensor Data Initialization
    i0[i, a] = 0.0;
    t1[i, a] = 1.0;
    t2[i, j, a, b] = 1.0;
    v[i, a, j, b] = 1.0;

    #Tensor Contraction
    i0[i, a] = v[c, d, m, n] * t2[i, n, a, d] * t1[m, c];
}
```

Chain of tensor contractions **without** multi-operand optimizations

```
$BUILD_DIR/bin/comet  \
    -lower-ta-to-ttgt \
    -opt-bestperm-ttgt \
    -opt-transpose     \
    -opt-matvec \
    -opt-matmul-tiling \
    $BENCH_DIR/ccsd_t1_11.ta \
    &>  $BENCH_IRs/ccsd_t1_11_case3_loop.mlir
```

Execution time: 5.847369 TIME

Testcase 5: SpMM in COO

```
def main() {
    #IndexLabel Declarations
    IndexLabelDynamic [a] = [?];
    IndexLabelDynamic [b] = [?];
    IndexLabel [c] = [32];

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

    #Tensor Data Initialization
    A[a, b] = read_from_file();
    B[b, c] = 1.0;
    C[a, c] = 0.0;

    #Tensor Contraction
    C[a, c] = A[a, b] * B[b, c];
}
```

Sparse kernel generation in COO

```
export SPARSE_FILE_NAME=${PWD}/../inputs/bcsstk17.mtx
$BUILD_DIR/bin/comet  \
    -lower-all-ta-to-loops \
    $BENCH_DIR/SpMM_COO.ta \
    &>  $BENCH_IRs/SpMM_COO_loop.mlir
```

Execution time: 0.002564 TIME

Testcase 6: SpMM in CSR

```
def main() {
    #IndexLabel Declarations
    IndexLabelDynamic [a] = [?];
    IndexLabelDynamic [b] = [?];
    IndexLabel [c] = [32];

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

    #Tensor Data Initialization
    A[a, b] = read_from_file();
    B[b, c] = 1.0;
    C[a, c] = 0.0;

    #Tensor Contraction
    C[a, c] = A[a, b] * B[b, c];
}
```

Sparse kernel generation in CSR

```
export SPARSE_FILE_NAME=${PWD}/../inputs/bcsstk17.mtx

$BUILD_DIR/bin/comet  \
    -lower-all-ta-to-loops \
    $BENCH_DIR/SpMM_CSR.ta \
    &>  $BENCH_IRs/SpMM_CSR_loop.mlir
```

Execution time: 0.001404 TIME

1.82x Speedup

Testcase 7: SpMM in DCSR

```
def main() {
    #IndexLabel Declarations
    IndexLabelDynamic [a] = [?];
    IndexLabelDynamic [b] = [?];
    IndexLabel [c] = [32];

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

    #Tensor Data Initialization
    A[a, b] = read_from_file();
    B[b, c] = 1.0;
    C[a, c] = 0.0;

    #Tensor Contraction
    C[a, c] = A[a, b] * B[b, c];
}
```

Sparse kernel generation in DCSR

```
export SPARSE_FILE_NAME=${PWD}/../inputs/bcsstk17.mtx

$BUILD_DIR/bin/comet  \
    -lower-all-ta-to-loops \
    $BENCH_DIR/SpMM_DCSR.ta \
    &>  $BENCH_IRs/SpMM_DCSR_loop.mlir
```

Execution time: 0.001260 TIME

2.03x Speedup

Pacific Northwest
NATIONAL LABORATORY

Demo: Testcase 5-6-7