DCTACO: A Data Centric Tensor Algebra Compiler

ANONYMOUS AUTHOR(S)*

Sparse tensor algebra plays a critical role in scientific computing, enabling efficient memory usage and computation by focusing on non-zero elements. While existing solutions like TACO provide a flexible framework for sparse tensor computations, challenges remain in handling higher-order tensors and optimizing disjunctive operations. This paper presents a novel compiler framework that integrates TACO's sparse iteration space with DaCe's Stateful DataFlow Graph (SDFG) intermediate representation. By introducing a canonical form for tensor expressions, we simplify lattice representations, enhance parallelism, and improve code generation. Our approach leverages the data-centric optimizations of DaCe, generating efficient computational kernels for sparse tensor algebra. Experimental evaluations on synthetic and real-world datasets demonstrate performance improvements of up to 50× over TACO in specific scenarios, particularly for operations involving disjunctive merges. This work advances the state of the art in sparse tensor compilers, laying the foundation for scalable and efficient handling of complex tensor computations.

CCS Concepts: • **Software and its engineering → Source code generation**.

Additional Key Words and Phrases: tensors, tensor algebra, data flow languages, sparse data structures, canonical form, iteration lattices, parallelism, HPC, code generation

1 INTRODUCTION

Sparse tensor algebra is fundamental in scientific computation, where its primary advantage lies in storing only non-zero elements of a tensor, thereby conserving memory and reducing unnecessary computations [1, 7, 8, 13, 16]. Since computations involving sparse tensors require specialized handling, various techniques and libraries have been developed to achieve optimal performance. Most of these efforts, however, have focused on two-dimensional tensors—specifically, sparse matrices [5, 20–22]. Extending efficient computation to higher-order tensors, particularly those with various storage formats, introduces additional complexities that sparse tensor algebra frameworks aim to address [10, 26, 28, 29, 31].

The primary challenge in sparse tensor algebra stems from the tensor storage format. Numerous data structures have been developed to represent sparse tensors, with the choice of structure often depending on how non-zero elements are distributed within the tensor. For instance, if the majority of non-zero elements are clustered around the diagonal of a matrix, the Diagonal (DIA) format is the most efficient choice. Taking into account the generality of real-world datasets, hardware support, front-end programming complexity, and compiler optimization, TACO [10] allows users to compose both dense and sparse dimensions. This flexibility enables users to choose the most appropriate storage format based on their specific scenario.

However, compressed storage formats can make tensor element access expensive, as index arrays must first be consulted to determine element locations. To improve the efficiency of element access and computation, TACO extends the dense iteration space concept to include sparse tensors [11]. The core idea is to construct a sparse iteration space and use lattice structures to represent the set of points in the space that need to be computed. Despite these improvements, computations over compressed tensors remain memory-bound, as each element access typically requires indirect accesses through metadata (e.g. `poi` and `idx` arrays in CSR formats). As a result, optimizing memory movement becomes crucial for improving performance in sparse tensor computations.

Data-Centric Parallel Programming (DaCe) is a parallel programming framework designed to decouple domain science from performance optimization [2]. At the heart of DaCe is its Stateful DataFlow multiGraph (SDFG) Intermediate Representation (IR), which can be generated from high-level languages such as Python, MATLAB and TensorFlow. DaCe also provides a builder

API that allows users to design custom front-ends. Unlike other dataflow IR models, such as the Program Dependence Graph (PDG) [4], which still focus on control dependencies, DaCe's SDFG revolves around the concept of stateful dataflow, fostering concurrency and enabling optimization over data-dependent operations [2].

This paper presents a novel tensor algebra compiler that integrates TACO's sparse iteration space with DaCe's stateful dataflow model. By leveraging DaCe's data-movement optimizations and generating SDFG IR from TACO's tensor algebra representation, our compiler is capable of optimizing a broad spectrum of tensor computations — from dense to highly sparse. This combined approach not only enhances performance but also simplifies the optimization process for applications requiring complex tensor operations, as DaCe inherently supports a wide range of optimizations.

1.1 key contributions

The contributions of this paper are:

(1) An approach to designing a domain-specific compiler based on SDFG IR for sparse tensor algebra.
(2) Proposed anonical form of tensor expressions to further develop TACO's lattice representation, easing code generation and pormote performance
(3) An performance evaluation of the automatically generated code with the proposed compiler. The comparison, based on generated sparse tensors and two real world tensors, shows that this work significantly outperforms the state-of-the-art TACO compiler in disjunctive operation.

2 BACKGROUND

2.1 Data-Centric Parallel Programming and SDFG IR

Data-Centric Parallel Programming (DaCe) is a framework that facilitates the separation of domain expertise from performance engineering. It allows programs written in high-level languages such as Python (with NumPy), MATLAB, and others to be transformed into Stateful Dataflow Multigraphs (SDFGs), an intermediate representation (IR) that decouples the architecture-specific programming paradigm from the underlying scientific computation. The responsibility of performance engineers is then to apply domain-specific optimizations that target this IR, allowing for the fine-tuning of the program for different hardware architectures without altering the original scientific code.

The SDFG IR is a graph-based intermediate representation in which nodes represent either accesses to data or computations, while the edges denote data movement or dependencies. One of the distinguishing features of the SDFG IR is its explicit handling of memory movement. Specifically, data access is represented through access nodes, and the reading and writing of data are specified by edges called *memlets*. Each memlet is annotated with precise information about the subset of data being accessed, ensuring that the flow of data between computations is clearly defined. Computations themselves are encapsulated within *tasklet* nodes, which only interact with data through these memlets, enforcing a strict separation between computation and data movement.

SDFGs also support higher-level constructs, such as loops and conditionals. Loops are represented using scope nodes, where the loop body is a nested SDFG, allowing for hierarchical representation of control flow. These nodes, along with others, are grouped within *states*, and the overall control flow of the program is represented by edges between these states, forming a state machine. This enables complex control flow while preserving dataflow-centric optimization opportunities. An example of this representation is shown in Fig. 1, which illustrates the SDFG for a Sparse Matrix-Vector (SpMV) multiplication.

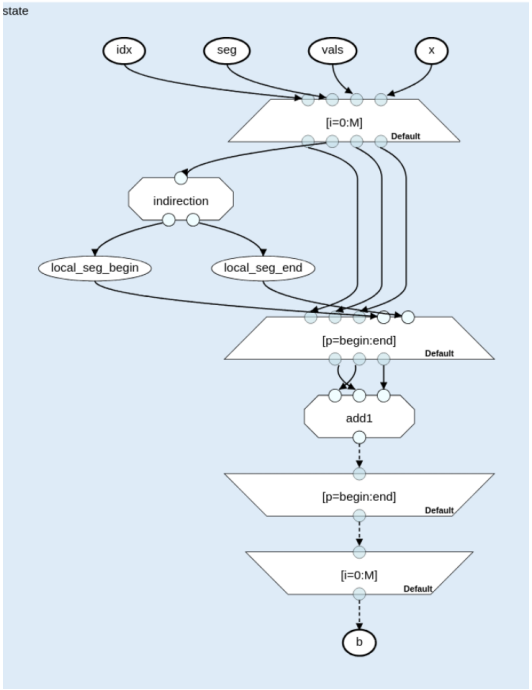


Fig. 1. SDFG representation of SpMV multiplication, where matrix is stroed in CSR format

2.2 The TACO Compiler

The Tensor Algebra Compiler (TACO) is a compiler-based library designed to handle tensor algebra expressions involving both sparse and dense tensors. There exist many formats supporting sparse tensor computation, most prevalently used are CSR/CSC, which represent a matrix using compressed one-dimensional arrays, and the Doubly Compressed Sparse Row (DCSR) format, which compresses a matrix in both dimensions.

TACO models tensors using a coordinate tree abstraction, where each level of the tree corresponds to a specific dimension of the tensor. In this structure, internal nodes represent coordinates within that dimension, while sub-trees represent sub-tensors of the original tensor. The entire tree, including its sub-trees, collectively forms a representation of the tensor. By splitting the tree at the first level, each resulting sub-tree represents a vector (i.e., a one-dimensional tensor), and each leaf corresponds to a scalar (i.e., a zero-dimensional tensor). Each leaf node in the tree has a unique path from the root, which defines the full coordinate of the corresponding element in the tensor. The removal of an element is conceptually equivalent to deleting its corresponding leaf node and any parent nodes that have no remaining children after the removal.

This coordinate tree abstraction provides the basis for TACO's ability to flexibly combine dimension orderings and compress individual tensor dimensions independently of others. Users can define how each tensor dimension is organized within the tree structure. Compression occurs when, upon splitting the tree at a given level, sub-trees containing only zero elements are removed, thereby conserving memory. This abstraction offers substantial representational flexibility, accommodating a wide variety of sparse tensor storage formats.

Beyond the coordinate tree abstraction, TACO introduces a domain-specific language (DSL) known as a Tensor Index Notation as shown in Fig. 2. This DSL provides a formalism for specifying

tensor computations, with a grammar structured around tensor operations, including assignments, reductions, and arithmetic expressions. The TACO compiler translates these tensor operations into highly optimized computational kernels, capable of executing efficiently on CPUs, GPUs, and distributed architectures.

$$
\begin{aligned}
\text{assignment} &::= \text{access } \text{"} = \text{"} \text{ expr} \\
\text{expr} &::= \text{reduction} \\
&\mid \text{operator} \\
&\mid \text{access} \\
&\mid \text{literal} \\
&\mid \text{"(" expr ")"} \\
\text{reduction} &::= \sum_{\text{indices}} \text{expr} \\
\text{operator} &::= \text{"} - \text{"} \text{ expr expr} \\
&\mid \text{"} + \text{"} \text{ expr expr} \\
&\mid \text{"} - \text{"} \text{ expr expr expr} \\
\text{access} &::= \text{tensor indices} \\
\text{indices} &::= \text{index}^*
\end{aligned}
$$

Fig. 2. Tensor Index Notation Expression Grammar[11]

TACO's code generation mechanism builds upon the concept of sparse iteration spaces, which generalizes the polyhedral model typically used for dense tensor algebra. This model represents affine loop nests as hyperrectangular iteration spaces. However, sparse computations introduce additional complexity due to compression, requiring the lookup of elements across compressed dimensions via index arrays (such as the `poi` and `idx` arrays in the CSR format). Moreover, fused tensor accesses in tensor expressions introduce expensive out-of-bounds access checks. To address these challenges, TACO introduces the notion of an iteration graph and a lattice, which are later generalized into the Iteration Lattice model. This advanced model efficiently handles the complexities of iterating over sparse tensor data structures whilst minimizing unnecessary computations [11].

3 THE TACO SYSTEM

Since our approach is originated from TACO, adopts the same front-end input, data structures, and amended it into our own compilation processes. This section will outlines core mechanisms of TACO, and Section. 4 will show how we advanceses it.

3.1 Tensor Algebra Expression

The input algebra expressions are specified using tensor index notation, as shown in Figure 2. This notation, widely used in tensor algebra systems [10, 24, 27], defines how each element in the result tensor is computed from elements in the operand tensors, independently of how the operands are accessed or how results are computed and stored.

The index notation is then used to generate an iteration graph, which, in combination with the tensor format, produces a merge lattice. These three components—iteration graph, merge lattice,

and tensor format—form what is known as TACO's IR. Although this IR has been extended with concepts such as the concrete index notation [9], the focus of this work is on data movement optimization, so we retain the original TACO model for tensor computation.

3.2 Tensor Storage Format

To store a tensor in memory, two components are required: the *data*, which consists of the tensor's elements, and the *metadata*, which describes how the elements are organized in memory. This metadata enables efficient access to specific elements via their coordinates, i.e., it provides a mapping from coordinates to elements.

A tensor is a multi-dimensional array of numerical values, with each value indexed by a tuple of coordinates corresponding to its position in each dimension. A tensor's shape is a tuple representing the length of each dimension. For a tensor with shape $(l_1, l_2, ..., l_k)$, where l_i is the length of the i-th dimension, the indices of elements in the i-th dimension satisfy $0 \leq idx_i < l_i$. Each element can be located by a tuple of k indices, $(idx_1, idx_2, ..., idx_k)$, which uniquely defines its position in the tensor.

In TACO's model, a tensor can also be represented as a *coordinate tree* [10], where each level of the tree corresponds to a dimension of the tensor. Nodes in the tree represent indices, with the root node representing the tensor as a whole (indexed by 0 in an outer dimension, e.g., l_{k+1}). A path from the root to a leaf node forms a coordinate, and each leaf corresponds to an element at that coordinate.

For dense tensors, all elements are stored in a contiguous memory space, represented by a one-dimensional *value* array. Assuming the tensor dimensions are ordered as $d_1, d_2, ..., d_k$ with lengths $l_1, l_2, ..., l_k$, the element at $(i_1, i_2, ..., i_k)$ is stored at position $i_1 \times \prod_{j=1}^{k} + i_2 \times \prod_{j=2}^{k} + ...$. Roughly speaking, the shape of the tensor and the ordering of dimensions together form the metadata, which dictates the spatial arrangement of elements in the value array.

While dense formats allow random access to elements, they require storing all elements, including zeros. To save space, sparse storage formats omit certain elements by effectively "deleting" corresponding leaves in the coordinate tree. However, this disrupts the straightforward mapping between coordinates and the spatial ordering of elements in the value array.

TACO's storage format combines dense and sparse levels [10]. Each level of the coordinate tree is associated with metadata: for dense levels, this is simply the length of the dimension, while for sparse levels, it consists of `pos` and `idx` arrays. If a level is sparse, nodes corresponding to zeros (and their subtrees) are deleted. If the last level has m nodes, a `pos` array of length $m + 1$ is used to segment the `idx` array, where each segment contains the indices of a node's children.

Sparse storage formats restrict random access because compression removes elements, breaking the spatial ordering. Consequently, dimensions above a sparse level must be accessed before accessing the sparse level, while dense dimensions can still be accessed directly. As shown in Figure 1, accessing elements in a sparse tensor requires an indirection step, creating dependencies on the sparse levels.

3.3 The Iteration Graph and Merge Lattice

TACO's IR consists of two main components: the iteration graph and the merge lattice, which have been further extended into the iteration lattice and concrete index notation [11]. Our approach adopts TACO's IR design for generating the SDFG IR.

The **Iteration Graph** guides the order of iteration through the multi-dimensional coordinate space of the tensor. The graph is built around index variables, which correspond to the tensor's dimensions (or levels in the coordinate tree). Traversing the coordinate tree is equivalent to iterating

over the index variables that define each level. Tensor element access is dependent on the ordering of dense and sparse levels. The iteration space is traversed in a "forest" ordering, where (1) parent nodes are always visited before their children, and (2) there are no circular dependencies, ensuring the iteration graph forms a directed acyclic graph (DAG).

The **Iteration Lattice** is a partially ordered set of levels used to abstract the iteration space, particularly when merging multiple dimensions. TACO's merge lattice is based on the generalization of the two-way merge algorithm [12]. The tensor index expression's abstract syntax tree (AST) is traversed in post-order to generate a lattice for each leaf node. Each lattice contains a set of levels, which are then merged to form the final lattice. Along with the levels, each lattice element is associated with a sub-expression, indicating the expression to be considered during code generation for the corresponding levels.

3.4 Disjunctive Merge

The concept of a merge lattice is motivated by disjunctive merges, which are introduced by disjunctive operations such as addition [10]. However, the original TACO paper [10] does not fully address the disjunctive merge in cases where mixed compressed and dense dimensions are involved. Specifically, the paper lacks instructions for handling the movement of iterators for dense levels, such as lines 14 and 18 in Figure 3, which are crucial for disjunctive operations.

Although TACO generates correct code for single-dimensional tensors involving mixed-mode disjunctive operations, challenges arise when dealing with higher-dimensional tensor algebra expressions such as result$(i, j) = A(i, j) + B(i, j)$, where A is Dense, Sparse, and both the result tensor and B are Dense, Sparse as shown in 3. In such cases, TACO produces segmentation faults. Most errors happen in disjunctive merge of several dense and sparse dimensions. Next section will illustrate how we overcome the defact.

```
1  int32_t i157 = 0;
2  int32_t i157A98 = A981_pos[0];
3  int32_t pA981_end = A981_pos[1];
4
5  while (i157A98 < pA981_end) {
6    int32_t i157A980 = A981_crd[i157A98];
7    if (i157A980 == i157) {
8      A154_vals[i157] = A98_vals[i157A98] + A65_vals[i157];
9    }
10   else {
11     A154_vals[i157] = A65_vals[i157];
12   }
13   i157A98 += (int32_t)(i157A980 == i157);
14   i157++;//Iterator over dense level
15 }
16 while (i157 < A651_dimension && i157 >= 0) {
17   A154_vals[i157] = A65_vals[i157];
18   i157++;//Iterator over dense level
19 }
```

Fig. 3. TACO code of $y(i) = (x\_temp(i) + x(i))$, where x_temp is sparse and x is compressed

4 CANONICAL FORM OF TENSOR EXPRESSIONS

The TACO system employs lattices to represent the iteration space for tensor computations. This approach is motivated by the need to handle disjunctive operations, such as addition, which can produce non-zero elements beyond just the overlapping areas of the iteration space. In TACO, each sub-lattice is represented by a loop, and every element in a sub-lattice is presented with an if-condition. However, this design can lead to increasingly complex loop conditions, an abundance of

```
    Tensor<double> A = read("example.mtx", csr);
    Tensor<double> B({A.getDimension(0), A.getDimension(1)}, dens);
    for (int i = 0; i < B.getDimension(0); ++i) {
        for (int j = 0; j < B.getDimension(1); ++j) {
            B.insert({i,j}, unif(gen));
        }
    }
    B.pack();
    int dim1 = A.getDimension(1);
    Tensor<double> result({A.getDimension(0),A.getDimension(1)}, dense);
    IndexVar i, j;
    result(i,j) = A(i,j) + B(i,j);
```

Fig. 4. The kernel of $result(i, j) = A(i, j) + B(i, j)$ generated by TACO, where A is Dense,Sparse and $result$ and B are Dense,Sparse

if-statements, and an iteration structure that complicates code correctness, as the actual computation within the loop body remains minimal. In this section, we introduce a method inspired by lattice theory and tensor expressions to simplify the lattice structure, reduce complexity, and expose greater parallelism potential.

4.1 Drawbacks of Two-Way Merging

We observed that the use of if-conditions within loops incurs a performance penalty, especially in sparse iteration spaces. Consider the example of the PLUS3 kernel for compressed sparse fiber (CSF) tensors, illustrated in Fig. 5a. Appendix A provides the code generated by TACO for this example. It is often more efficient to sum operand tensors separately, restructuring them into a form such as Result = ((Result + A) + B) + C. For instance, sparse vector addition by two separate loops (one for each vector added to the result vector) typically outperforms the two-way merge algorithm. Example code for this approach is shown in Appendix B.

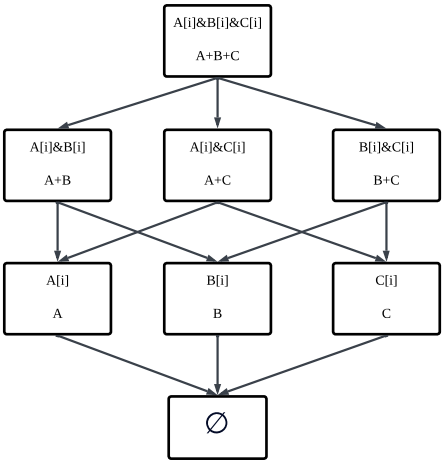
Originally designed for sorting tasks rather than arithmetic, the two-way merge approach does not account for the accumulative nature of disjunctive operations like addition. TACO's model builds lattices recursively for each operand tensor, where addition operations especially generate lattices that contain all original lattice points as well as newly created ones. For example, Fig. 5a demonstrates the lattice for a PLUS3 operation, with the TACO-generated code available in Appendix A. TACO uses the two-way merge algorithm [12] to iterate over each dimension in a single pass.

However, this approach poses challenges beyond performance. As lattice structures grow more complex, they comprise a combinatorial increase in points, making it error-prone to partition sub-lattices and generate code with appropriate boundary checks. Additionally, the number of if-statements rises, leading to increased time spent on condition evaluations. When multiple compressed and dense levels are present in an expression, TACO may generate incorrect code, as shown in Section 3.4.

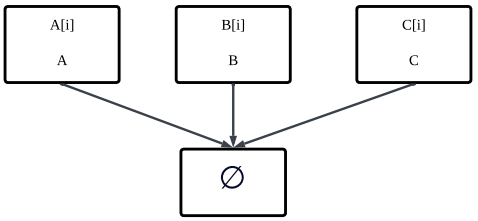
4.2 Canonical Form

To enable more effective code generation, tensor expressions should first be fully expanded into a **Canonical Form**, akin to a polynomial, by applying the distributive law. A tensor expression can be represented as a tree, where each leaf represents a tensor. Our method divides this tree into a forest, where each tree contains only multiplication terms, also known as monomials. Similar to TACO, we generate an SDFG intermediate representation (IR) for each monomial and then sum these intermediate results to obtain the final result.

From the perspective of the iteration lattice, each monomial contains only one if-statement, as it has no smaller elements in the lattice. This reduces the number of boundary checks needed, although each dimension may be iterated multiple times, once for each monomial. This approach effectively keeps only the join-irreducible elements in the iteration lattice, computing partial results that are then summed to yield the final result. Fig. 5b shows the reduced iteration lattice, and Fig. 6 illustrates the transformation to canonical form. Within each monomial, tensors are ordered from left to right according to their final dimension in the iteration graph, ensuring that expressions are generated as soon as they become available at a given level in the graph.



(a) Iteration lattice of the PLUS3 operation over third-order sparse tensors



(b) Reduced iteration lattice for the PLUS3 operation over third-order sparse tensors

Instead of relying on the two-way merge algorithm, we apply the distributive law to expand tensor expressions, generating a lattice for each monomial and then summing them. Although this may result in some data access overhead, it reduces time spent on condition evaluations and improves branch prediction.

For dense tensors, this approach might increase runtime slightly, but in sparse tensors, the search consists of many conditional (indirect) accesses. The two-way merge algorithm introduces conditions proportional to the number of elements in the lattice.

Additionally, this transformation enhances parallelism. As shown in Section 6.1, when the outer level contains more than one sparse level, TACO cannot generate parallel loops because it requires one iterator for each sparse dimension, limiting parallelism. By converting expressions

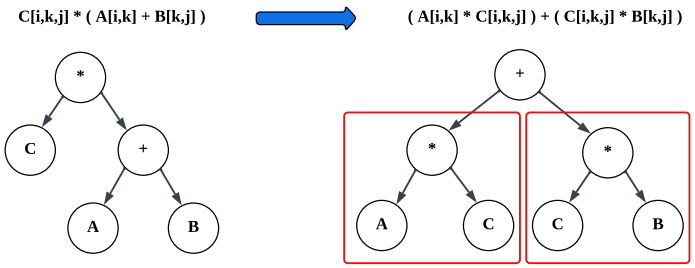


Fig. 6. Transformation to canonical form: $C[i,j] = C[i,j,k] \times (A[i,k] + B[k,i])$

into Canonical Form, we can generate parallel loops for each monomial. Each monomial can then be processed in the same manner as TACO, but without the complexities of disjunctive merges, allowing for easier and more efficient code generation.

5 LOWERING TO SDFG IR

In this section, we describe how to transform TACO IR into SDFG IR. The resulting SDFG is a nested structure where each level corresponds to an index variable, and the parent-child relationships in the iteration graph indicate that one SDFG is directly nested within another. The number of index variables determines the depth of this nesting.

The benefit of SDFG IR is that it can better represent data flow across sparse levels, easing the prosess of generaiting code, and leverage the data flow optimization of Dace framework. Given canonical form of tensor expression we can recursively generate SDFG IR for each monomial, and connect them through sunmmation.

Due to the differences between TACO IR, which focuses on sparse iteration spaces, and SDFG IR, which emphasizes stateful dataflow, the transformation can result in unnecessarily complex SDFG structures. Therefore, optimization is required post-transformation, which will be discussed in Section **??**. Figure 7 illustrates the intuitive correspondence between SDFG constructs and TACO IR components.

5.1 SDFG IR Generation Algorithm

Canonical form of tensor expressions is used to generate the SDFG (Stateful DataFlow Graph), allowing us to handle each monomial separately. Fig. 8 illustrates the monomial from the perspective of lowering. The key concept is to treat the result tensor and operand tensors independently. When reaching the final dimension of the result tensor, a temporary reduction variable is emitted to indicate reduction behavior. When accessing the last dimension of an operand tensor, which enables retrieval of values from the `vals` array, the tensor access is evaluated and multiplied by any previously computed values, if applicable. Finally, each SDFG intermediate representation (IR) is connected to the result tensor to aggregate the final output.

Fig. 9 presents the pseudo-code for transforming TACO IR into SDFG IR. The general approach is to recursively add SDFG components to construct the SDFG IR for the tensor algebra kernel. Fig. 10 demonstrates the generated MTTKRP (Matricized Tensor Times Khatri-Rao Product) kernel, where matrices A, C, and D are dense, and tensor B is three-dimensional, with all dimensions stored in a compressed format. The result tensor is initialized to zero before computation.

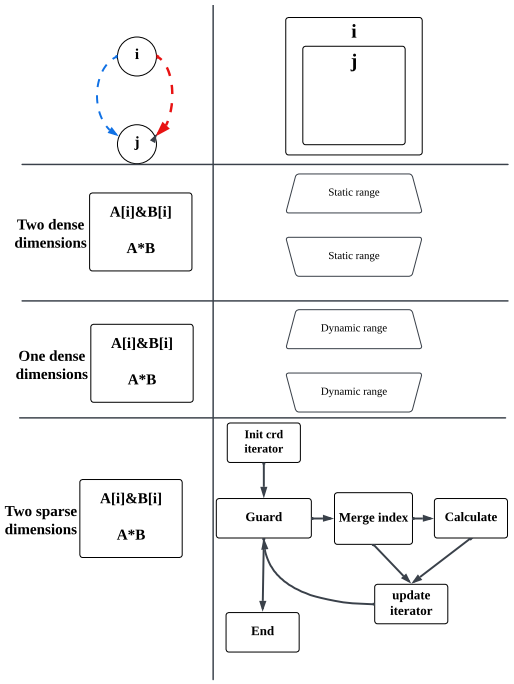


Fig. 7. Intuitive correspondence between SDFG and TACO IR

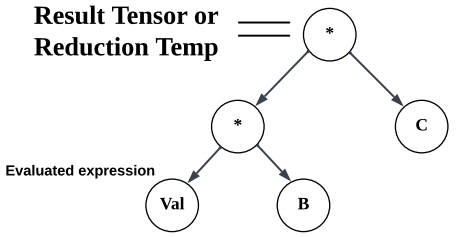


Fig. 8. Canonical form of a tensor expression, ensuring that available tensor access is evaluated to a single value, thereby avoiding redundant calculations.

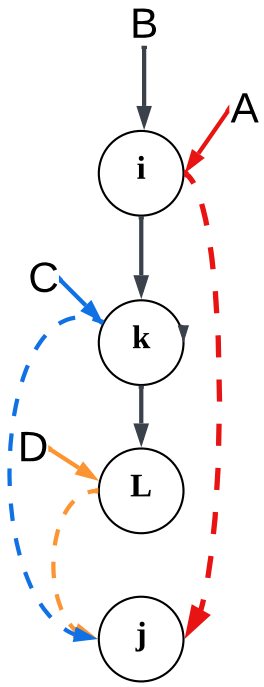
The transformation process is recursive, starting with the index notation expression and the first index variable in the iteration graph. Initially, a merge lattice is constructed to model the iteration space. Fig. 10a illustrates the generated MTTKRP kernel in which $A_{ij} = B_{ikl} \cdot D_{lj} \cdot C_{kj}$. Here, matrices A, C, and D are dense, and B is a three-dimensional tensor stored in a compressed format. The result tensor is initialized to zero before computation.

If the current index variable represents the last level in a tensor's coordinate tree, computational code is generated. If it is the final level for the result tensor, but additional iteration is required for the operands, a reduction step is added, writing the reduced result to the output tensor. When the level is not the last for any tensor, the algorithm recurses with the sub-expression and the

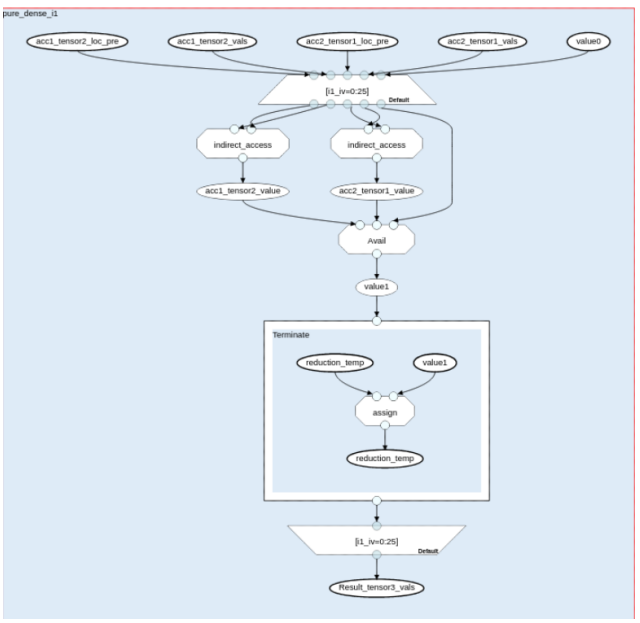
next index variable in the iteration graph. Naming conventions ensure that each tensor and index variable is uniquely identified, enabling clear associations between position variables and tensor accesses.

```
for each monomial:
    emit_SDFG(index-expr, monomial, iv) # iv is the index variable
    if monomial has reduction LHS and evaluated RHS:
        emit_terminate()

    # Declare data container for each dimension
    add_data_container(monomial) for dimensions after iv in iteration graph

    # Get number of sparse levels in this dimension
    n = get_num_sparse_levels(monomial, iv)

    if n == 0:
        emit_static_map()
    elif n == 1:
        emit_dynamic_map()
    else:
        emit_while_loop()

emit_sum(all monomials)
```

Fig. 9. Recursive algorithm to transform the canonical form of a tensor expression into SDFG IR



(a) Iteration graph for MTTKRP



(b) Inner level of generated SDFG for MTTKRP (full graph too large to display)

Fig. 10. Recursive transformation of TACO IR to SDFG IR, applied to MTTKRP where B is DCSR, C and D are dense.

5.2 Generating a Computation Tasklet

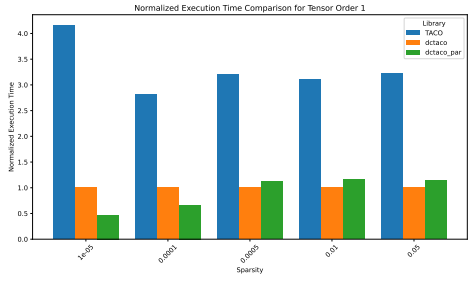
In SDFG IR, computations are encapsulated within tasklets, which separate computation from data movement. The programmer specifies the data available for computation, and in tensor algebra kernels, computations are performed on tensors at their final levels, where full coordinates are accessible. Each operand tensor must be accessed at the recursion depth corresponding to its final level. If the result tensor is available at a particular level but the operand tensors require further iteration, a reduction is generated, indicating that the result tensor has fewer dimensions than the full iteration space.

Following TACO's design [10], if the recursion reaches the last free variable (i.e., the last variable indexing the result tensor), a write operation is generated for the result tensor. If further nested recursions remain, a reduction construct is added. If recursion occurs before reaching the last free variable, intermediate sub-expressions are computed and passed to subsequent recursions, minimizing redundant calculations.

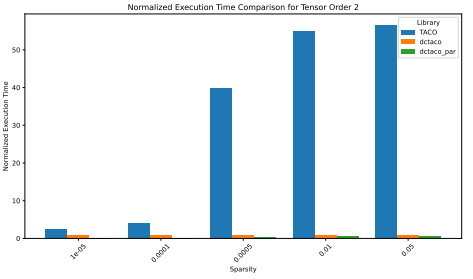
6 EVALUATION

All experiments were conducted on an Intel Xeon Platinum 8570 processor with 56 cores and 256 GB of memory, running RedHat Enterprise 8.8. Our implementation was compared against TACO at commit `2b8ece4`. We implemented our compiler using the DaCe Builder API as an extension of the DaCe ecosystem. Each experiment was run with 5 warmup rounds, and the arithmetic mean execution time was calculated over 20 benchmark rounds.

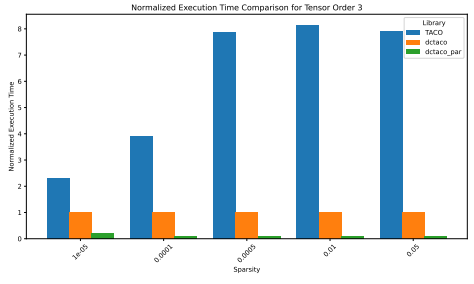
6.1 PLUS3 Kernel



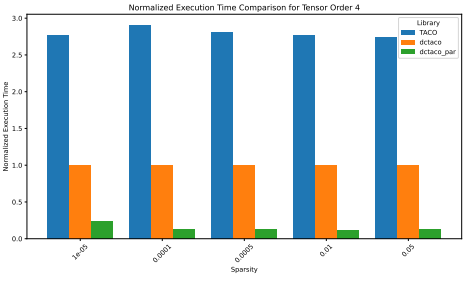
(a) PLUS3 on a [1,000,000]-dimensional sparse vector



(b) PLUS3 on a [1,000 × 10,000] DCSR matrix



(c) PLUS3 on a [1,000 × 1,000 × 1,000] CSF tensor



(d) PLUS3 on a [100 × 100 × 100 × 100] CSF tensor

Fig. 11. Normalized execution time of PLUS3 kernels over tensors compressed across all dimensions.

To evaluate the effectiveness of replacing the two-way merge algorithm, we used the PLUS3 kernel, which operates over three CSF tensors generated with varying sparsity levels. Because the kernel involves three sparse tensors, TACO cannot generate parallel for-loops. In contrast, our approach generates OpenMP parallel for-loops. Additionally, even in serial execution, our method demonstrates performance improvements ranging from 2x to over 50x compared to TACO, depending on the input tensor. This is particularly evident for matrices, as shown in Fig. 11b.

The observed speedup primarily arises from the removal of if-conditions. When the average dimension length is short, the first condition is more likely to evaluate to true, avoiding unnecessary computations. This characteristic makes our approach scalable with input size since larger tensors (in every dimension) lead to a higher likelihood of the two-way merge algorithm evaluating more conditions. The same principle applies to parallel execution: the larger the tensor, the greater the performance gains achieved through parallelism.

6.2 High-Order Tensor Algebra

We also evaluated several high-order tensor algebra kernels on datasets from the FROSTT benchmark suite [23]. Table 1 shows the performance results for two datasets: NELL-2 and NELL-1. For NELL-2, our approach (dctaco) performs comparably to TACO, while for NELL-1, the performance is worse. The difference arises from the dataset characteristics: NELL-2 has dimensions $12,092 \times 9,184 \times 28,818$, whereas NELL-1 is significantly larger with dimensions $2,902,330 \times 2,143,368 \times 25,495,389$.

The performance degradation on NELL-1 may stem from memory system limitations. The DaCe framework uses NumPy arrays to interact with Python, which could introduce a performance penalty. Identifying the precise cause is an area for future investigation.

Table 1. Performance of high-order tensor algebra kernels. All third-order tensors are stored in CSF format, while other operand and result tensors are dense.

	NELL-2		NELL-1	
	TACO	dctaco	TACO	dctaco
TTV	1	1.11	1	1.90
MTTKRP	1	0.97	1	1.71
MTTKRP-par	1	1.16	1	1.28
INNERPROD	1	1.18	1	1.74

7 RELATED WORK

Sparse Tensor Compiler and Optimization Techniques. Compiler advancements have significantly contributed to tensor algebra computations, with notable tools designed for both dense and sparse tensor operations. In the dense domain, TCE[6] and COMET[19] specialize in tensor contraction optimizations, particularly for quantum chemistry and computational chemistry.

In contrast, sparse tensor computations present unique challenges due to irregular access patterns and non-affine loops, making traditional optimizations less effective. TACO[10] is a leading sparse compiler generating efficient code for diverse storage formats across multi-core and GPU architectures. However, TACO's architecture-specific code generation often requires significant adaptation efforts to support new architectures. To address these limitations, a multi-level IR-based compiler is proposed to facilitate extensibility and interoperability across domains, supporting both serial and parallel code generation with enhanced portability for future architectures.

Sparse tensor contraction optimizations have also evolved through frameworks like the Sparse Polyhedral Framework and systems such as Athena[17] and Sparta[18], which optimize sparse operations by restructuring data layout and maximizing parallelism. However, many existing compilers, including TACO, COMET, and Sparsifier, face limitations in handling complex nested loops and exploring extensive search spaces. Emerging tools like SparseTIR[30], SparseLNR[3], and ReACT[32] offer fused sparse loop generation but are still limited in flexibility for arbitrary loop nests with branching.

Dataflow and Data-Centric Optimization in Intermediate Representations. Many intermediate representations (IRs) combine dataflow and control flow within graph structures, such as LLVM[15] IR's control-flow graphs in SSA form and Program Dependence Graphs (PDGs)[4], which connect data dependencies and control flow for traditional architectures. Unlike these, State-Diagram Flow Graphs (SDFGs)[2] use explicit dataflow-driven state machines, supporting reconfigurable hardware and enabling more precise data movement tracking through detailed memlet definitions.

Data-centric optimizations are a core feature in systems like Halide and CHiLL, which prioritize data flow in transformations. Higher-level optimizations are also applied in HPVM[14], Lift[25], and other frameworks, which focus on complex data movement patterns and enable GPU mapping and parallelism. While SDFGs can convert to SSA and PDGs, their native concurrency features are often lost, showcasing SDFGs' unique role in encapsulating multi-level data dependencies and concurrency that other IRs lack.

8 CONCLUSIONS

In this work, we introduced a domain-specific compiler that combines TACO's tensor algebra capabilities with DaCe's data-centric optimization framework. By formulating tensor expressions in a canonical form, we addressed the limitations of traditional lattice-based methods, simplifying code generation and enabling better exploitation of parallelism. Our approach leverages DaCe's Stateful DataFlow Graphs (SDFGs) to improve performance across a range of tensor computations, particularly in scenarios involving disjunctive operations.

The experimental results validate the effectiveness of our method, showing significant performance gains over TACO on synthetic and real-world datasets. However, challenges remain, particularly in handling extreme sparsity levels, as seen with the NELL-1 dataset, where memory system limitations impact performance. Future work will explore enhancements to the interaction between the DaCe framework and Python-based data structures, as well as strategies to further optimize memory movement in highly sparse tensors.

By bridging the gap between sparse tensor algebra and data-centric optimization, this work provides a robust framework for scalable tensor computation, with applications spanning scientific computing, machine learning, and beyond.

REFERENCES

[1] Nathan Bell, Steven Dalton, and Luke N Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.

[2] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 81, 14 pages. https://doi.org/10.1145/3295500.3356173

[3] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. 2022. SparseLNR: accelerating sparse tensor computations using loop nest restructuring. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 15,

14 pages. https://doi.org/10.1145/3524059.3532386

[4] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[5] Lukas Gianinazzi, Alexandros Nikolaos Ziogas, Langwen Huang, Piotr Luczynski, Saleh Ashkboosh, Florian Scheidl, Armon Carigiet, Chio Ge, Nabil Abubaker, Maciej Besta, et al. 2024. Arrow Matrix Decomposition: A Novel Approach for Communication-Efficient Sparse Matrix Multiplication. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 404–416.

[6] So Hirata. 2003. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A* 107, 46 (2003), 9887–9897.

[7] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra.* SIAM.

[8] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[9] Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 180–192. https://doi.org/10.1109/CGO.2019.8661185

[10] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[11] Fredrik Berg Kjølstad. 2020. *Sparse tensor algebra compilation.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[12] DE Knuth. 1998. The art of computer programming: Sorting and searching, vol. 3 2nd ed addison wesley longman publishing co. *Redwood City, CA* (1998), 458–478.

[13] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.

[14] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. Hpvm: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 68–80.

[15] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[16] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2015. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 806–814.

[17] Jiawen Liu, Dong Li, Roberto Gioiosa, and Jiajia Li. 2021. Athena: high-performance sparse tensor contraction sequence on heterogeneous memory. In *Proceedings of the 35th ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 190–202. https://doi.org/10.1145/3447818.3460355

[18] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: high-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 318–333. https://doi.org/10.1145/3437801.3441581

[19] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2021. COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry. arXiv:2102.06827 [cs.MS] https://arxiv.org/abs/2102.06827

[20] Nvidia. 2024. cuSPARSE. https://docs.nvidia.com/cuda/cusparse/. Accessed: October 15, 2024.

[21] Nvidia. 2024. cuSPARSE. https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-2/sparse-blas-functionality.html. Accessed: October 15, 2024.

[22] SciPy. 2024. scipy.sparse. https://docs.scipy.org/doc/scipy/reference/sparse.html. Accessed: October 15, 2024.

[23] Shaden Smith, Jee W Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The formidable repository of open sparse tensors and tools.

[24] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190.

[25] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/2784731.2784754

[26] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

[27] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[28] Jaeyeon Won, Changwan Hong, Charith Mendis, Joel Emer, and Saman Amarasinghe. 2023. Unified Convolution Framework: A compiler-based approach to support sparse convolutions. In *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5. Curan, 666–679. https://proceedings.mlsys.org/paper_files/paper/2023/file/ccf7262fb986e4367ccd3903960c57a0-Paper-mlsys2023.pdf

[29] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. https://doi.org/10.1145/3582016.3582047

[30] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. https://doi.org/10.1145/3582016.3582047

[31] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 213–232. https://www.usenix.org/conference/osdi22/presentation/zheng-ningxin

[32] Tong Zhou, Ruiqin Tian, Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, and Vivek Sarkar. 2023. ReACT: Redundancy-Aware Code Generation for Tensor Expressions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3559009.3569685

A PLUS3 CODE GENERATED BY TACO

```
      while (i122A0 < pA01_end && i122A59 < pA591_end) {
    int32_t i122A00 = A01_crd[i122A0];
    int32_t i122A590 = A591_crd[i122A59];
    int32_t i122 = TACO_MIN(i122A00,i122A590);
    if (i122A00 == i122 && i122A590 == i122) {
      int32_t i123A0 = A02_pos[i122A0];
      int32_t pA02_end = A02_pos[(i122A0 + 1)];
      int32_t i123A59 = A592_pos[i122A59];
      int32_t pA592_end = A592_pos[(i122A59 + 1)];

      while (i123A0 < pA02_end && i123A59 < pA592_end) {
        int32_t i123A00 = A02_crd[i123A0];
        int32_t i123A590 = A592_crd[i123A59];
        int32_t i123 = TACO_MIN(i123A00,i123A590);
        if (i123A00 == i123 && i123A590 == i123) {
          int32_t i123A63 = i122 * A632_dimension + i123;
          A63_vals[i123A63] = A0_vals[i123A0] + A59_vals[i123A59];
        }
        else if (i123A00 == i123) {
          int32_t i123A63 = i122 * A632_dimension + i123;
          A63_vals[i123A63] = A0_vals[i123A0];
        }
        else {
          int32_t i123A63 = i122 * A632_dimension + i123;
          A63_vals[i123A63] = A59_vals[i123A59];
        }
        i123A0 += (int32_t)(i123A00 == i123);
        i123A59 += (int32_t)(i123A590 == i123);
      }
      while (i123A0 < pA02_end) {
        int32_t i123 = A02_crd[i123A0];
        int32_t i123A63 = i122 * A632_dimension + i123;
        A63_vals[i123A63] = A0_vals[i123A0];
        i123A0++;
      }
      while (i123A59 < pA592_end) {
        int32_t i123 = A592_crd[i123A59];
        int32_t i123A63 = i122 * A632_dimension + i123;
        A63_vals[i123A63] = A59_vals[i123A59];
```

```
      i123A59++;
    }
  }
  else if (i122A00 == i122) {
    for (int32_t i123A0 = A02_pos[i122A0]; i123A0 < A02_pos[(i122A0 + 1)]; i123A0++) {
      int32_t i123 = A02_crd[i123A0];
      int32_t i123A63 = i122 * A632_dimension + i123;
      A63_vals[i123A63] = A0_vals[i123A0];
    }
  }
  else {
    for (int32_t i123A59 = A592_pos[i122A59]; i123A59 < A592_pos[(i122A59 + 1)]; i123A59++) {
      int32_t i123 = A592_crd[i123A59];
      int32_t i123A63 = i122 * A632_dimension + i123;
      A63_vals[i123A63] = A59_vals[i123A59];
    }
  }
  i122A0 += (int32_t)(i122A00 == i122);
  i122A59 += (int32_t)(i122A590 == i122);
}
while (i122A0 < pA01_end) {
  int32_t i122 = A01_crd[i122A0];
  for (int32_t i123A0 = A02_pos[i122A0]; i123A0 < A02_pos[(i122A0 + 1)]; i123A0++) {
    int32_t i123 = A02_crd[i123A0];
    int32_t i123A63 = i122 * A632_dimension + i123;
    A63_vals[i123A63] = A0_vals[i123A0];
  }
  i122A0++;
}
while (i122A59 < pA591_end) {
  int32_t i122 = A591_crd[i122A59];
  for (int32_t i123A59 = A592_pos[i122A59]; i123A59 < A592_pos[(i122A59 + 1)]; i123A59++) {
    int32_t i123 = A592_crd[i123A59];
    int32_t i123A63 = i122 * A632_dimension + i123;
    A63_vals[i123A63] = A59_vals[i123A59];
  }
  i122A59++;
}
```

B TWO-WAY MERGE ON SPARSE VECTOR ADDITION

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define MIN(_a,_b) ((_a) < (_b) ? (_a) : (_b))

#define VECTOR_SIZE 100000000 // Adjust size as needed
#define SPARSITY 0.0001 // Adjust sparsity as needed
#define EPSILON 1e-6 // Tolerance for floating-point comparison

// Initialize a dense vector with all zeros
void initialize_zero(float *vec, int size) {
    for (int i = 0; i < size; i++) {
        vec[i] = 0.0f;
    }
}

// Initialize a sparse vector with random values and sparsity
void initialize_sparse(int *indices, float *values, int size, int *non_zero_count) {
    *non_zero_count = 0;
    for (int i = 0; i < size; i++) {
        if ((float)rand() / RAND_MAX < SPARSITY) {
            indices[*non_zero_count] = i;
            values[*non_zero_count] = ((float)rand() / RAND_MAX) * 10.0f; // Random value
            (*non_zero_count)++;
        }
    }
}
```

```
// Validation function to check if two arrays are approximately equal
int validate_results(const float *a1, const float *a2, int size) {
    for (int i = 0; i < size; i++) {
        if (fabs(a1[i] - a2[i]) > EPSILON) {
            printf("Mismatch at index %d: a1 = %f, a2 = %f\n", i, a1[i], a2[i]);
            return 0; // Return false if there is any mismatch
        }
    }
    return 1; // Return true if all elements are within tolerance
}

// Two-way merge
void two_way_merge(float *a, const int *b_indices, const float *b_values, int b_nnz,
                const int *c_indices, const float *c_values, int c_nnz,
                const int *d_indices, const float *d_values, int d_nnz) {
    int b_it_loc=0;
    int c_it_loc=0;
    int d_it_loc=0;
    while(b_it_loc<b_nnz && c_it_loc<c_nnz && d_it_loc<d_nnz){
        int index_b = b_indices[b_it_loc];
        int index_c = c_indices[c_it_loc];
        int index_d = d_indices[d_it_loc];

        int iv=MIN(index_b, MIN(index_d, index_c));
        if(iv==index_c&&iv==index_d&&iv==index_b){
            a[iv]=b_values[b_it_loc] + c_values[c_it_loc]*d_values[d_it_loc];
        }
        else if(iv==index_b){
            a[iv]=b_values[b_it_loc];
        }
        else if(iv==index_c&&iv==index_d){
            a[iv]=c_values[c_it_loc]*d_values[d_it_loc];
        }

        if(iv==index_b) b_it_loc++;
        if(iv==index_c) c_it_loc++;
        if(iv==index_d) d_it_loc++;

    }

    while(c_it_loc<c_nnz && d_it_loc<d_nnz){
        int index_c = c_indices[c_it_loc];
        int index_d = d_indices[d_it_loc];

        int iv=MIN(index_d, index_c);
        if(iv==index_d&&iv==index_c){
            a[iv]= c_values[c_it_loc]*d_values[d_it_loc];
        }
        if(iv==index_c) c_it_loc++;
        if(iv==index_d) d_it_loc++;

    }
    while(b_it_loc<b_nnz ){
        int index_b = b_indices[b_it_loc];
        a[index_b]=b_values[b_it_loc];
        b_it_loc++;

    }

}

// Two-loop approach
void two_loop(float *a, const int *b_indices, const float *b_values, int b_nnz,
             const int *c_indices, const float *c_values, int c_nnz,
             const int *d_indices, const float *d_values, int d_nnz) {

    int b_it_loc=0;
    int c_it_loc=0;
    int d_it_loc=0;
    while(c_it_loc<c_nnz && d_it_loc<d_nnz){
        int index_c = c_indices[c_it_loc];
```

```
        int index_d = d_indices[d_it_loc];

        int iv= MIN(index_d, index_c);
        if(iv==index_c&&iv==index_d){
            a[iv]=c_values[c_it_loc]*d_values[d_it_loc];
        }

        if(iv==index_c) c_it_loc++;
        if(iv==index_d) d_it_loc++;
    }

    while(b_it_loc<b_nnz ){
        int index_b = b_indices[b_it_loc];
        a[index_b]+=b_values[b_it_loc];
        b_it_loc++;

    }
}

// Benchmark function
double benchmark(void (*func)(float*, const int*, const float*, int, const int*, const float*, int, const int
      *, const float*, int),
                 float *a, const int *b_indices, const float *b_values, int b_nnz,
                 const int *c_indices, const float *c_values, int c_nnz,
                 const int *d_indices, const float *d_values, int d_nnz) {
    clock_t start = clock();
    func(a, b_indices, b_values, b_nnz, c_indices, c_values, c_nnz, d_indices, d_values, d_nnz);
    printf("%f/n", a[10]);
    return (double)(clock() - start) / CLOCKS_PER_SEC;
}

int main() {
    // Allocate memory
    float *a1 = (float*)calloc(VECTOR_SIZE, sizeof(float));
    float *a2 = (float*)calloc(VECTOR_SIZE, sizeof(float));
    initialize_zero(a1, VECTOR_SIZE);
    initialize_zero(a2, VECTOR_SIZE);

    int *b_indices = (int*)malloc(VECTOR_SIZE * sizeof(int));
    float *b_values = (float*)malloc(VECTOR_SIZE * sizeof(float));
    int b_nnz;

    int *c_indices = (int*)malloc(VECTOR_SIZE * sizeof(int));
    float *c_values = (float*)malloc(VECTOR_SIZE * sizeof(float));
    int c_nnz;

    int *d_indices = (int*)malloc(VECTOR_SIZE * sizeof(int));
    float *d_values = (float*)malloc(VECTOR_SIZE * sizeof(float));
    int d_nnz;

    // Initialize sparse vectors
    initialize_sparse(b_indices, b_values, VECTOR_SIZE, &b_nnz);
    initialize_sparse(c_indices, c_values, VECTOR_SIZE, &c_nnz);
    initialize_sparse(d_indices, d_values, VECTOR_SIZE, &d_nnz);

    // Run benchmarks
    double time_two_way_merge = benchmark(two_way_merge, a1, b_indices, b_values, b_nnz, c_indices, c_values,
          c_nnz, d_indices, d_values, d_nnz);
    double time_two_loop = benchmark(two_loop, a2, b_indices, b_values, b_nnz, c_indices, c_values, c_nnz,
          d_indices, d_values, d_nnz);

    printf("two-way-merge time: %f seconds\n", time_two_way_merge);
    printf("Two-loop time: %f seconds\n", time_two_loop);

        // Validate results
    if (validate_results(a1, a2, VECTOR_SIZE)) {
        printf("Validation successful: Results match!\n");
    } else {
        printf("Validation failed: Results do not match.\n");
    }
    // Cleanup
```

```
    free(a1);
    free(a2);
    free(b_indices);
    free(b_values);
    free(c_indices);
    free(c_values);
    free(d_indices);
    free(d_values);

    return 0;
}
```

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009