CoNST: Code Generator for Sparse Tensor Networks

SAURABH RAJE, Kahlert School of Computing, University of Utah, Salt Lake City, United States
YUFAN XU, Kahlert School of Computing, University of Utah, Salt Lake City, United States
ATANAS ROUNTEV, Computer Science and Engineering, Ohio State University, Columbus, United States
EDWARD F. VALEEV, Chemistry, Virginia Tech, Blacksburg, United States
P. SADAYAPPAN, Kahlert School of Computing, University of Utah, Salt Lake City, United States

Sparse tensor networks represent contractions over multiple sparse tensors. Tensor contractions are higher-order analogs of matrix multiplication. Tensor networks arise commonly in many domains of scientific computing and data science. Such networks are typically computed using a tree of binary contractions. Several critical inter-dependent aspects must be considered in the generation of efficient code for a contraction tree, including sparse tensor layout mode order, loop fusion to reduce intermediate tensors, and the mutual dependence of loop order, mode order, and contraction order. We propose CoNST, a novel approach that considers these factors in an integrated manner using a single formulation. Our approach creates a constraint system that encodes these decisions and their interdependence, while aiming to produce reduced-order intermediate tensors via fusion. The constraint system is solved by the Z3 SMT solver and the result is used to create the desired fused loop structure and tensor mode layouts for the entire contraction tree. This structure is lowered to the IR of the TACO compiler, which is then used to generate executable code. Our experimental evaluation demonstrates significant performance improvements over current state-of-the-art sparse tensor compiler/library alternatives.

CCS Concepts: • **Software and its engineering → Source code generation**; **Domain specific languages**;

Additional Key Words and Phrases: Sparse tensors, tensor networks, tensor layout, loop fusion

ACM Reference Format:
Saurabh Raje, Yufan Xu, Atanas Rountev, Edward F. Valeev, and P. Sadayappan. 2024. CoNST: Code Generator for Sparse Tensor Networks. *ACM Trans. Arch. Code Optim.* 21, 4, Article 82 (November 2024), 24 pages. https://doi.org/10.1145/3689342

1 Introduction

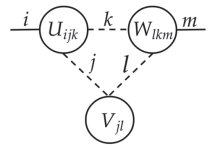
This article describes CoNST, a **Co**de generator for **N**etworks of **S**parse **T**ensors. A tensor network expresses a collection of *tensor contractions* over a set of tensors. Tensor contractions

This work was supported in part by the U.S. National Science Foundation through awards 2009007, 2216903, 2217081, and 2217154.
Authors' Contact Information: Saurabh Raje, Kahlert School of Computing, University of Utah, Salt Lake City, Utah, United States; e-mail: saurabh.raje@utah.edu; Yufan Xu, Kahlert School of Computing, University of Utah, Salt Lake City, Utah, United States; e-mail: yf.xu@utah.edu; Atanas Rountev, Computer Science and Engineering, Ohio State University, Columbus, Ohio, United States; e-mail: rountev.1@osu.edu; Edward F. Valeev, Chemistry, Virginia Tech, Blacksburg, Virginia, United States; e-mail: efv@vt.edu; P. Sadayappan, Kahlert School of Computing, University of Utah, Salt Lake City, Utah, United States; e-mail: saday@cs.utah.edu.

ACM 1544-3566/2024/11-ART82
https://doi.org/10.1145/3689342

are higher-order analogs of matrix–matrix multiplication. For example, the binary contraction $Y_{ijlm} = U_{ijk} \times W_{klm}$ represents the computation $\forall i, j, l, m : Y_{ijlm} = \sum_k U_{ijk} \times W_{klm}$. Multi-tensor product expressions, e.g., $Z_{im} = U_{ijk} \times V_{jl} \times W_{klm}$, arise commonly in many domains of scientific computing and data science (e.g., high-order models in quantum chemistry [35], tensor decomposition schemes [20]). They involve multiple tensors and multiple summation indices, e.g., $\forall i, m : Z_{im} = \sum_{j,k,l} U_{ijk} \times V_{jl} \times W_{klm}$.

These multi-tensor products are also referred to as *tensor networks*, represented with a node for every tensor instance and edges representing variables that index the various tensors. The figure on the right illustrates this representation. As explained later in Section 2, such a network is typically computed efficiently using a tree of binary contractions.



Tensor network

Considerable prior research has been directed at the optimization of dense tensor contractions [1, 17, 22, 26, 29, 30, 36, 37, 43] and the optimization of tensor networks where the component tensors are dense [10, 15, 32]. A few efforts have also addressed the optimization of tensor networks with sparse tensors under some restrictions [12, 16, 19, 27, 51]. However, optimization and effective code generation for arbitrary sparse tensor networks remain an unsolved challenge.

A fundamental difficulty in developing efficient sparse versions of tensor computations in comparison to the corresponding dense versions is the fact that some compact representation such as **Compressed Sparse Fiber (CSF**, [40, 41], detailed in Section 2) must be used to represent the non-zero elements, with the implication that arbitrary slices of a multi-dimensional tensor cannot be efficiently extracted. In contrast, with dense tensors, arbitrary elements or contiguous slices along any combination of tensor modes can be easily and efficiently accessed.[1] Therefore, while code generation for the application of an arbitrary combination of loop transformations like tiling, permutation, and fusion is quite straightforward for the dense case, the same is not true for optimization and code generation for a collection of sparse tensor contractions.

Loop fusion transforms a sequence of perfectly nested loops into an imperfectly nested loop, where a set of one or more outermost loops with exactly matching loop bounds from each of the loop nests is pulled out and made common surrounding loops for a sequence of lower-dimensional loop nests containing the non-common loops. Consider the simplest case of loop fusion across a sequence of two perfectly nested loops, where the first loop nest produces an array that is consumed by the second loop nest. The cache reuse distance (defined as the number of distinct data elements accessed between two successive accesses to a given data element) for the fused version of the code can be significantly lower than the unfused version. This is because lower-dimensional slices of the produced/consumed array (corresponding to fixed values of the fused loop iterators) are produced/consumed in temporal proximity with the fused version, whereas all accesses to the entire array happen for the first loop-nest before any accesses from the second loop-nest for the unfused code, resulting in much larger reuse distances.

The above benefit of improved data locality and reuse in cache for fused producer/consumer loops applies to both dense and sparse tensor contractions. However, for the sparse context, an additional benefit accrues from loop fusion. When a set of common surrounding loops between a producer loop-nest and a consumer loop-nest is fused, it is not necessary to allocate the full space for the temporary array that is produced/consumed, but only as much as needed for lower-dimensional slices corresponding to fixed values for the fused loop iterators. This is because space used for a previous slice (corresponding to some fixed values of the fused loop iterators) can be

[1]We use "mode" to refer to a tensor dimension; an order-n tensor has n modes. Terminology details are presented in Section 2.

reused for the next slice. Further, if a sufficient number of loops are fused and the size of the full product data space for the lower-dimensional slice is small, a dense representation can be used instead of an explicit sparse representation for the slices of the intermediate temporary tensor between the producer and consumer statement, thereby lowering data access overheads [18].

Although a few efforts have been directed toward compiler optimization of sparse matrix and tensor computations [7, 12, 18, 19, 24, 25, 44, 45, 50], the current state of the art does not adequately address a number of critical inter-dependent aspects in the generation of efficient fused code for a given tree of sparse binary contractions.

Sparse tensor layout mode order. We focus on the widely used CSF format, which is commonly used for efficient sparse tensor computations. Since CSF uses a nested representation with n levels for a tensor of order n, efficient access is only feasible for some groupings of non-zero elements by traversing the hierarchical nesting structure. Selecting the nesting of the n modes of a tensor is a key factor for achieving high performance. Prior efforts in compiler optimization and code generation for sparse computations have not explored the impact of the choice of CSF nested layout mode order on the performance of contraction tree evaluation.

Loop fusion to reduce intermediate tensors. The temporary intermediate tensors that correspond to inner nodes of the contraction tree could be much larger than the input and output tensors of the network. By fusing common loops of the nested loops that produce and consume an intermediate tensor, the size of that tensor can be reduced significantly (as illustrated by an example in Section 2). A reduction of the size of an intermediate tensor can enable significant reduction in the number of cache misses if the reduced intermediate can fit in cache but the intermediate without loop fusion does not. Further, a dense representation of the intermediate becomes feasible, which further improves performance due to the reduced cost of tensor element accesses [18].

Inter-dependence between loop order, mode order, and contraction order. In addition to selecting the layout mode order for each tensor in the contraction tree, code generation needs to select a legal loop fusion structure to implement the contractions from the tree. Such a fused structure depends on the order of surrounding loops for each contraction, on the order in which the contractions are executed, and on the choice of layout mode order. No existing work considers the space of these inter-related choices in a systematic and general manner.

Our solution. We propose CoNST, a novel approach that considers the above factors in an integrated manner using a single formulation. This formulation encodes several inter-related goals. First, for each contraction, we ensure that the order of loops that surround it (in the fused loop nest) is compatible with the layout mode order of all tensors that participate in the contraction. This allows for efficient traversal of non-zero elements of these tensors. Second, we produce a valid topological sort of the contraction tree (i.e., each producer contraction appears before the corresponding consumer contraction). Third, the surrounding loops for each producer–consumer pair allow for valid fusion—and not only for this pair, but also for all other contractions that appear between the pair in the topological sort. Finally, the resulting fusion allows for significant reduction of intermediates: specifically, all intermediate tensors are guaranteed to be of order at most l, where l is a small constant limit (e.g., $l = 1$) given as a parameter to our tool.

To find a solution that satisfies these goals, we formulate a constraint system in which constraint variables are used to encode all relevant choices: order of surrounding loops, order of tensor layout modes, and topological order of contractions. The system is then solved by the Z3 SMT solver [11] and the result is used to create the desired fused loop structure and tensor mode layouts for the entire contraction tree. This structure is lowered to the IR of the **Tensor Algebra Compiler (TACO)** [19], which is then used to generate the final executable code. The main contributions of CoNST are:

— We design a novel constraint-based approach for encoding the space of possible fused loop structures and tensor CSF layouts, with the goal of reducing the order of intermediate tensors. This is the first work that proposes such a general integrated view of code generation for sparse tensor contraction trees.
— We develop an approach to translate the constraint solution to the concrete index notation IR [18] of the TACO compiler.
— We perform extensive experimental comparison with the three most closely related systems: TACO [19], SparseLNR [12], and Sparta [25]. Using a variety of benchmarks from quantum chemistry and tensor decomposition, we demonstrate significant performance improvements over this state of the art.

2 Background and Overview

2.1 Tensor Networks

We first describe the abstract specification of a tensor network. Such a specification can be lowered to many possible code implementations. Examples of such implementations are also given below.

Sparse tensors. A tensor T of order n is defined by a sequence $\langle d_0, \ldots, d_{n-1} \rangle$ of *modes*. Each mode d_k denotes a set of index values: $d_k = \{x \in \mathbb{N} : 0 \leq x < N_k\}$, where N_k is the *mode extent*. Note that the numbering of modes from 0 to $n-1$ is purely for notational purposes and does not imply any particular concrete data layout representation; deciding on such a layout is one of the goals of our work, as described later. For a sparse tensor T, its non-zero structure is defined by some subset $nz(T)$ of the Cartesian product $d_0 \times d_2 \times \cdots \times d_{n-1}$. All and only non-zero elements of T have coordinates that are in $nz(T)$. Each $(x_0, x_1, \ldots) \in nz(T)$ is associated with a non-zero value $T(x_0, x_1, \ldots) \in \mathbb{R}$.

The tensor expressions described below use *tensor references*. A reference to an order-n tensor T is defined by a sequence $\langle i_0, \ldots, i_{n-1} \rangle$ of distinct *iteration indices* ("indices" for short). Such a reference will be denoted by $T_{i_0 i_1 \ldots}$. Each index i_k is mapped to the corresponding mode d_k of T and denotes the values defined by that mode: $i_k = \{x \in \mathbb{N} : 0 \leq x < N_k\}$. The same index may appear in several tensor references, for the same tensor or for different ones. In all such occurrences, the index denotes the same set of index values. For example, an expression discussed shortly contains tensor references X_{ijqr}, A_{ipq}, and B_{jpr}. As an illustration, index j appears in two of these references and is mapped to mode 1 of X and mode 0 of B (and thus both modes have the same extent).

CSF representation. Our work focuses on sparse tensors represented in the widely used CSF format [40, 41]. CSF organizes a sparse tensor as a tree, defined by some permutation of modes $d_0, \ldots, d_{n-1}$. This order of modes defines the CSF layout and must be decided when creating a concrete implementation of the computation. The internal nodes of the tree store the indices of non-zero elements in the corresponding mode. The leaves of the tree store the non-zero values. An auxiliary root node connects the entire structure. Using the sparse format abstractions introduced by Chou et al. [9], the outermost mode is dense (i.e., all index values are represented), while the remaining ones are compressed (i.e., only index values with corresponding non-zero elements are represented).[2]

Figure 1 illustrates the CSF representation for an order-4 sparse tensor. When the abstract specification of a tensor expression (or equivalently, of a tensor network) is lowered to a concrete implementation, both tensors and tensor references are instantiated to specific representations. For example, suppose we have a tensor A with modes d_0, d_1, and d_2, and a reference A_{ipq} appears

[2] Our approach also trivially applies to a CSF variation in which the outermost mode is compressed.

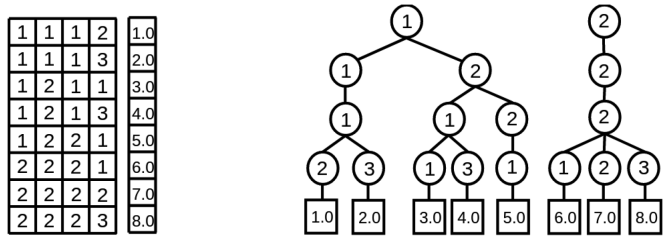
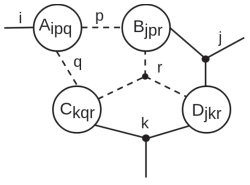


Fig. 1. The CSF format for representing an order-4 sparse tensor in memory. The table on the left shows the indices of non-zero elements. The tree on the right shows the CSF representation (root node is not shown).



(a) Tensor network

```
R[*,*,*] = 0
for i
 for j
  for k
   for p
    for q
     for r
      R[i,j,k] += A[i,p,q]*B[j,p,r]*
                  C[k,q,r]*D[j,k,r]
```

(b) N-ary contraction

Fig. 2. Tensor network and code for N-ary contraction for expression $R_{ijk} = A_{ipq} \times B_{jpr} \times C_{kqr} \times D_{jkr}$.

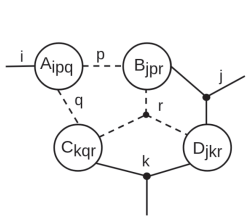
in the tensor network. One (of many) possible implementation is to order the modes as d_1, d_2, d_0 in outer-to-inner CSF order. The code references to the tensor would be consistent with this order; i.e., reference A_{ipq} becomes `A[p,q,i]` in the code implementation.

Tensor contractions and tensor networks. Consider tensors T, S, and R and a *binary contraction* $R_{i_0 i_1 \ldots} = T_{j_0 j_1 \ldots} \times S_{k_0 k_1 \ldots}$. Let In_T, In_S, and In_R denote the sets of indices appearing in each tensor reference, respectively. Any index $i \in In_R$ is an *external* index for this contraction. Any index $i \in (In_T \cup In_S) \setminus In_R$ is a *contraction* index for the contraction.

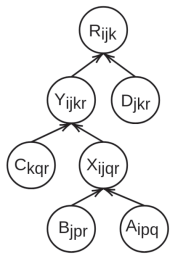
The non-zero structure of R is defined by the non-zero structure of T and S as follows: $(z_0, z_1, \ldots) \in nz(R)$ if and only if there exists at least one pair of tuples $(x_0, x_1, \ldots) \in nz(T)$ and $(y_0, y_1, \ldots) \in nz(S)$ such that for each index $i \in In_T \cup In_S \cup In_R$, the values corresponding to i in the three tuples (if present) are the same. For any $(z_0, z_1, \ldots) \in nz(R)$, the associated value $R(z_0, z_1, \ldots) \in \mathbb{R}$ is the sum of $T(x_0, x_1, \ldots) \times S(y_0, y_1, \ldots)$ for all such "matching" pairs of tuples $(x_0, x_1, \ldots) \in nz(T)$ and $(y_0, y_1, \ldots) \in nz(S)$. As a simple example, $R_{ij} = T_{ik} \times S_{kj}$ represents a standard matrix multiplication: for any $(a, b) \in nz(R)$ we have $R(a, b) = \sum_{\{c:(a,c)\in nz(T) \wedge (c,b)\in nz(S)\}} T(a, c) \times S(c, b)$.

A general (non-binary) contraction expression of the form $R_{\ldots} = T1_{\ldots} \times \cdots \times Tn_{\ldots}$ is defined similarly. Such an expression can be equivalently represented as a *tensor network*, with one vertex for each tensor reference in the expression, and a hyper-edge for every index. An example of a tensor network representing the tensor expression $R_{ijk} = A_{ipq} \times B_{jpr} \times C_{kqr} \times D_{jkr}$ is shown in Figure 2(a). Here dashed hyperedges are used to distinguish the *contraction* indices in the tensor expression (i.e., i, j, and k) from the *external* indices.

The direct computation of any tensor network (multi-tensor product expression) can be performed via a nested loop, with one loop corresponding to each index, and a single statement that mirrors the tensor expression. Figure 2(b) shows pseudo-code for such an N-ary contraction, where



(a) Tensor network



(b) Contraction tree

$$X_{ijqr} = A_{ipq} \times B_{jpr}$$
$$Y_{ijkr} = X_{ijqr} \times C_{kqr}$$
$$R_{ijk} = Y_{ijkr} \times D_{jkr}$$

(c) Binary contractions

Fig. 3. Contraction tree for a tensor network.

N is the number of operand tensors. Here and for later examples that apply in both the dense and sparse context, we often do not explicitly indicate loop bounds because the form will differ for the dense and sparse case. Note that the figure shows a specific code version with a concrete loop order (e.g., i in the outermost position) and tensor data layouts (e.g., j is the outermost CSF level of D). There are many possible choices for the loop order and the tensor layout. While the loops are straightforward in the dense case, for sparse CSF tensors the code is much more complex, and general techniques for such iteration have been developed [19].

For the dense case, the complexity of such an implementation is $O(M_i M_j M_k M_p M_q M_r)$, where M_x are the corresponding extents. By exploiting associativity and distributivity, the multi-term product can be rewritten as a sequence of binary contractions, with temporary intermediate tensors X and Y as shown in Figure 3(c). By using a *sequence of binary contractions* instead of an N-ary contraction, the complexity is reduced to $O(M_i M_j M_p M_q M_r + M_i M_j M_k M_q M_r + M_i M_j M_k M_r)$. If all tensor modes have the same extent M, the complexity reduces from $O(M^6)$ to $O(M^5)$.

There exist many different sequences of binary tensor contractions to compute a tensor network, with varying computational complexity. The problem of identifying an operation-optimal sequence of binary contractions for a multi-term product expression is NP-complete [8], but practically effective solutions have been developed for this problem [15, 32, 38]. We assume that one of these solutions has been applied to produce a binary contraction tree and consider the orthogonal problem of generating efficient code to implement that contraction tree.

2.2 Challenges and Overview of Solution

The problem we address in this article is the following: *Given a binary tensor contraction tree for a sparse tensor network, generate efficient code for its evaluation.*

Loop fusion and dimension reduction of intermediates. Figure 4(a) shows one possible code implementation for the contraction tree from Figure 3(b). Since identical loops over indices i and j exist in the loop code for all three binary contractions, we can fuse those loops to create the imperfectly nested loop structure in Figure 4(b). We note that the unfused code version in Figure 4(a) requires 4-D intermediate arrays $X[i, j, q, r]$ and $Y[i, j, k, r]$, but the fused code in Figure 4(b) can use 2-D intermediate arrays $X[q, r]$ and $Y[k, r]$. This is because i and j are common surrounding loops, thus producing and consuming 2-D slices of the 4-D intermediate arrays. The same space can be reused for a later slice since all values produced for a previous slice have been fully used.

In contrast to the dense case, with sparse tensor contractions a fundamental challenge is that of insertion of each additive contribution from the product of a pair of elements of the input tensors to the appropriate element of a sparse output tensor. The TACO compiler [19] defines a *workspaces* optimization [18] to address this challenge, where a dense multidimensional temporary array is used to assemble multidimensional slices of the output tensor during the contraction of sparse

```
X[*,*,*,*] = 0
Y[*,*,*,*] =0
R[*,*,*] = 0
for i,j,p,q,r
   X[i,j,q,r] += A[i,p,q]*B[j,p,r]
for i,j,k,q,r
   Y[i,j,k,r] += X[i,j,q,r]*C[k,q,r]
for i,j,k,r
   R[i,j,k] += Y[i,j,k,r]*D[j,k,r]
```

(a) Unfused sequence of contractions

```
R[*,*,*]=0
for i,j
  X[*,*]=0
  Y[*,*]=0
  for p,q,r
     X[q,r] += A[i,p,q]*B[j,p,r]
  for k,q,r
     Y[k,r] += X[q,r]*C[k,q,r]
  for k,r
     R[i,j,k] += Y[k,r]*D[j,k,r]
```

(b) Common loops i and j fused

Fig. 4. Reduction of dimensionality of intermediate tensors via loop fusion.

input tensors. By using a dense "workspace," very efficient $O(1)$ cost access to arbitrary elements in the slice is achieved for assembling the irregularly scattered contributions generated during the contraction. A significant consideration with the use of the dense workspaces is the space required: the extents of the workspace array must equal the extents of the corresponding modes of the sparse output tensor and thus can become excessive. By use of loop fusion between producer and consumer contractions to reduce the number of explicitly represented modes in intermediate tensors, we represent all intermediates as dense workspaces and thus make efficient use of TACO's workspaces optimization.

In addition to fusion, a critical factor for high performance is the compatibility between loop order and layout order. For sparse CSF tensors, efficient access to the non-zero elements is only feasible if the outer-to-inner order of nested loop indices in the code implementation is consistent with the layout order of tensor modes, in relation to the loop indices that index them. For example, the elements referenced by `A[i,p,q]` can be accessed efficiently only if i appears earlier than p (which itself appears earlier than q) in the loops surrounding this reference.

To summarize, given a binary contraction tree to implement a general sparse tensor contraction expression, three critical inter-related decisions affect the performance of the generated code:

- *Linear execution order of contractions:* The fusibility of loops between a producer contraction of an intermediate tensor and a subsequent consumer contraction is affected by the linear execution order of the contractions.
- *Loop permutation order for each contraction:* All surrounding loops of a contraction are fully permutable. The chosen permutation affects both the fusibility of loops across tensor contractions and the efficiency of access of non-zero elements of sparse tensors.
- *Mode layout order for each tensor:* The compatibility of the layout order of each tensor with the loop order of the surrounding loops is essential for efficient access.

These three decisions are inter-dependent. The linear execution order (i.e., the topological sort of the contraction tree) affects which loop fusion structures are possible. The order of loops for each contraction determines what fusion can be achieved, while also imposing constraints on the data layouts of tensors that appear in the contraction tree. In this article, we propose a novel integrated solution that considers these three decisions in a single formulation. Our approach creates a constraint system that encodes the space of possible decisions and their interdependence. This system is then solved using the Z3 SMT solver [11]. The solution is used to create a legal fused loop structure that reduces the size of intermediate tensors while ensuring the compatibility constraints described above.

Table 1. Comparison with State-of-the-art Systems for Sparse Tensor Computations

	TACO	SparseLNR	Sparta	CoNST (ours)
Loop fusion	✗	✓	✗	✓
Data layout selection	✗	✗	✗	✓
Schedule for contraction trees	✗	✗	✗	✓

To the best of our knowledge, this is the first work that takes such an integrated view and provides a general approach for code generation for arbitrary tensor contraction trees. Table 1 contrasts our work with the three most closely related state-of-the-art systems for sparse tensor computations described below. Our experimental evaluation presents comparisons with all three existing systems. Section 6 provides further details on these and other related efforts.

The CoNST system leverages, as its last stage, the code generator for sparse tensor computations in TACO [19]. The main focus of TACO is the generation of efficient code for N-ary contractions with arbitrarily complex tensor expressions. While TACO can be used to generate code for a sequence of binary sparse tensor contractions, it does not address optimizations like loop fusion across tensor contractions, tensor mode layout choice, or the choice of sequence of tensor contractions for a given contraction tree. In our experimental evaluation (Section 5), we show that code generated by CoNST achieves significant speedup over code directly generated by TACO.

SparseLNR [12] builds on TACO to implement loop fusion optimization. It takes a multi-term tensor product expression as input and generates fused loop code for a sequence of binary tensor contractions corresponding to the input tensor product expression. In our experimental evaluation, we compare the performance of code generated by SparseLNR with code generated by CoNST and demonstrate significant speedups.

Sparta [25] implements a library for efficient tensor contraction of arbitrary pairs of sparse tensors. Since a library is being created, this work does not address any optimizations like loop fusion across contractions, data layout choice for tensors, or the schedule of contractions for a contraction tree. We performed extensive experimentation to compare the performance of code generated by CoNST with the best performance among all valid tensor layout permutations for unfused sequences of contractions executed using Sparta. These experiments demonstrate very significant performance gains for CoNST.

3 Constraint-based Integrated Fusion and Data Layout Selection

Our approach aims to generate a concrete implementation of a given contraction tree by automatically determining (1) the order of modes in the data layout of each tensor and (2) a structure of fused loops that minimizes the order of intermediate tensors. We formulate a constraint system that answers the following question: For the given contraction tree, does there exist an implementation for which all intermediate tensors are of order at most l, for some given integer l? We first ask this question for $l = 1$. If the answer is positive, the constraint system solution is used to construct a code implementation for the contraction tree. If the answer is negative, we formulate and solve a constraint system for $l = 2$, seeking a solution in which all intermediates are at most 2-D matrices. This process continues until we find a solution. Note that a trivial solution without any fusion is guaranteed to exist for a sufficiently large value of l.

In each of these steps, we employ the Z3 SMT solver [11] to provide either (1) a negative answer ("the constraint system is unsatisfiable") or (2) a positive answer with a concrete constraint solution that defines the desired tensor layouts and loop structure. The generated constraints are based on quantifier-free integer difference logic. While in general the search space is exponential in

the number of contractions and the number of indices, our experience shows that Z3 solves the generated constraint systems with very practical running times (as detailed in Section 5).

3.1 Input and Output

The input to our approach is a set of contractions $\{C_0, C_1, \ldots, C_{m-1}\}$ organized in a contraction tree. Each leaf node corresponds to an input tensor reference, the root node corresponds to a result tensor reference, and every other node corresponds to an intermediate tensor reference. As an example, the contraction tree for $X_{ijqr} = A_{ipq} \times B_{jpr}; Y_{ijkr} = X_{ijqr} \times C_{kqr}; R_{ijk} = Y_{ijkr} \times D_{jkr}$ was shown earlier in Figure 3(b). Here A, B, and C are input tensors; X and Y are intermediate tensors; and R is the result tensor.

A naive implementation of a given tree would contain a sequence of perfectly nested loops (one loop nest per contraction), based on some valid topological sort order of tree nodes. For each contraction, the loop nest would be some permutation of the set of indices that appear in the tensor references, and the loop body would be a single assignment. For example, the loop nest for $X_{ijqr} = A_{ipq} \times B_{jpr}$ would contain loops for r, q, i, j, and p in some order.

As discussed earlier in Section 2, for any (unfused or fused) implementation, a fundamental constraint is that the order of surrounding loops must match the data layout order of modes in the CSF tensor representation. This is needed to allow for efficient iteration over the sparse representation. For example, consider reference A_{ipq}. Recall from the earlier discussion that each index is mapped to the corresponding mode of A: i is mapped to d_0, p is mapped to d_1, and q is mapped to d_2. A concrete implementation would select a particular order of d_0, d_1, and d_2 as the outer, middle, and inner level in the CSF representation, respectively. For example, suppose that this order is, from outer to inner, $\langle d_1, d_2, d_0 \rangle$. In the code implementation, the tensor reference would be `A[p,q,i]`. Efficient iteration over elements of A would require that the loop structure surrounding the reference matches this order: the p loop must appear before the q loop, which must appear before the i loop. The constraint-based approach described below incorporates such constraints for the loops that surround (in a fused code structure) each tensor reference from the contraction tree.

Each of the fused loop structures we would like to explore can be uniquely defined by (1) a topological sort order of the non-leaf nodes in the contraction tree and (2) for each such node, an ordering of the indices that appear in it. The index order for a node defines the order of loops that would surround the corresponding assignment in the fused loop nest. This order also defines the CSF layout order for the corresponding tensors.

```
for r,j
  for p,q,i
     X[q,i] += A[p,q,i]*B[r,j,p]
  for q,k,i
     Y[k,i] += X[q,i]*C[r,q,k]
  for k,i
     R[j,k,i] += Y[k,i]*D[r,j,k]
```

Fig. 5. Fused code structure.

For example, consider the code structure in Figure 5, which is derived from the solution of our constraint system for the running example. Here there is a single valid topological sort for the assignments. The ordering of surrounding loops for the assignments is $\langle r, j, p, q, i \rangle$, $\langle r, j, q, k, i \rangle$, and $\langle r, j, k, i \rangle$, respectively. The fusion of the common r and j loops allows X and Y to be reduced to 2-D tensors. The order of indices in all tensor references is consistent with the order of surrounding loops.

3.2 Constraint Formulation

The space of targeted code structures is encoded via constraints over integer-typed constraint variables, as described below.

3.2.1 Ordering of Assignments. For each contraction C_i, the position of the corresponding assignment relative to the other assignments in the code is encoded by a constraint variable ap_i (short for "assignment position for C_i") such that $0 \leq ap_i < m$, $ap_i \neq ap_k$ for all $k \neq i$, and $ap_i < ap_j$ if C_i is a child of C_j in the contraction tree. Here m is the number of contractions. The first two constraints guarantee uniqueness and appropriate range for all ap_i. The last constraint ensures a valid topological sort order.

Example. For the running example, we have ap_0 for $X_{ijqr} = A_{ipq} \times B_{jpr}$, ap_1 for $Y_{ijkr} = X_{ijqr} \times C_{kqr}$, and ap_2 for $R_{ijk} = Y_{ijkr} \times D_{jkr}$. For this contraction tree (Figure 3) the only valid topological sort is C_0, C_1, C_2 and thus the only possible solution is $ap_i = i$. In a more general tree, there may be multiple valid assignments of values to ap_i, each corresponding to one topological sort order.

3.2.2 Ordering of Tensor Modes. For each order-n tensor T that has references in the contraction tree, and each mode d_j of T ($0 \leq j < n$), we use a constraint variable $dp_{T,j}$ to encode the position of d_j in the CSF layout of the tensor. The following constraints are used: $0 \leq dp_{T,j} < n$ and $dp_{T,j} \neq dp_{T,j'}$ for all $j' \neq j$. Any constraint variable values that satisfy these constraints define a particular permutation of the modes of tensor T and thus a concrete CSF data layout.

Example. In the running example, A has three modes and thus three constraint variables $dp_{A,0}$, $dp_{A,1}$, and $dp_{A,2}$. In the code structure shown in Figure 5, abstract tensor reference A_{ipq} is mapped to concrete reference `A[p,q,i]`. This corresponds to the following assignment of values to the constraint variables: $dp_{A,0} = 2$, $dp_{A,1} = 0$, and $dp_{A,2} = 1$. Thus, the outermost level in the CSF representation corresponds to mode d_1 (indexed by p), the next CSF level corresponds to d_2 (indexed by q), and the inner CSF level corresponds to d_0 (indexed by i).

3.2.3 Ordering of Loops. Next, we consider constraints that encode the fused loop structure. For any contraction C_i, we need to encode the loop order of the loops surrounding the corresponding assignment. Let In_i be the set of indices that appear in C_i. For each $k \in In_i$, we define an integer constraint variable $lp_{i,k}$ (short for "loop position of index k for C_i"). These variables will encode a permutation of the elements of In_i, that is, a loop order for the loops surrounding the assignment for C_i. If $lp_{i,k}$ has a value of 0, index k will be the outermost loop surrounding the assignment. If the value is 1, the index will be the second-outermost loop, and so forth. To encode a permutation, for each $k \in In_i$ we have constraints $0 \leq lp_{i,k} < |In_i|$ and $lp_{i,k} \neq lp_{i,k'}$ for all $k' \in In_i \setminus \{k\}$.

Example. In the running example, for contraction $C_0 : X_{ijqr} = A_{ipq} \times B_{jpr}$ we have $In_0 = \{i, j, p, q, r\}$. For this contraction we will use constraint variables $lp_{0,i}$, $lp_{0,j}$, $lp_{0,p}$, $lp_{0,q}$, $lp_{0,r}$. In the code structure shown in Figure 5, the loop order for C_0 is $\langle r, j, p, q, i \rangle$. This order corresponds to a constraint solution in which $lp_{0,i} = 4$, $lp_{0,j} = 1$, $lp_{0,p} = 2$, $lp_{0,q} = 3$, and $lp_{0,r} = 0$.

3.2.4 Consistency between Mode Order and Loop Order. Next, we need to ensure that the order of loops defined by $lp_{i,k}$ is consistent with the order of modes for each tensor appearing in contraction C_i, as encoded by $dp_{T,j}$. Consider a reference to T appearing in contraction C_i. For each pair of modes d_j and $d_{j'}$ of T, let k and k' be the indices that correspond to these modes in the reference. The following constraint enforces the consistency between mode order and loop order:

$$(dp_{T,j} < dp_{T,j'}) \Rightarrow (lp_{i,k} < lp_{i,k'}).$$

Here $dp_{T,j} < dp_{T,j'}$ is true if and only if mode d_j appears earlier than mode $d_{j'}$ in the concrete CSF data layout of tensor T. If this is the case, we want to enforce that the index corresponding to d_j (i.e., k) appears earlier than the index corresponding to $d_{j'}$ (i.e., k') in the loop order of loops surrounding the assignment for C_i. As discussed earlier, this constraint ensures that the order of iteration defined by the loop order allows an efficient traversal of the CSF data structure for

T. Such constraints are introduced for all input tensors. For intermediates that are represented through dense workspaces, such constraints are not necessary. In our implementation, we use dense workspaces for all intermediates.

Example. Consider reference A_{ipq} from the running example and the pair of modes d_0 and d_2, with corresponding indices i and q. The relationship between variables $dp_{A,0}$ (for d_0), $dp_{A,2}$ (for d_2), $lp_{0,i}$ (for i), and $lp_{0,q}$ (for q) is captured by the following two constraints:

$$(dp_{A,0} < dp_{A,2}) \Rightarrow (lp_{0,i} < lp_{0,q}) \qquad (dp_{A,2} < dp_{A,0}) \Rightarrow (lp_{0,q} < lp_{0,i}).$$

As described earlier, in the constraint solution we have $dp_{A,0} = 2$, $dp_{A,2} = 1$, $lp_{0,i} = 4$, and $lp_{0,q} = 3$. Of course, these values satisfy both constraints.

3.2.5 Producer–consumer Pairs. Finally, we consider every pair of contractions C_i, C_j such that C_i is a child of C_j in the contraction tree. In this case C_i produces a reference to a tensor T that is then consumed by C_j. Let n be the order of T. Our goal is to identify a loop fusion structure that reduces the order of this intermediate tensor T to be some $n' \leq l$ for a given parameter l. Recall that in our overall scheme, we first define a constraint system with $l = 1$. If this system cannot be satisfied, we define a new system with $l = 2$, and so forth.

Let In_T be the set of indices that appear in the reference to T. We define constraints that include $lp_{i,k}$ (for the producer C_i) and $lp_{j,k}$ (for the consumer C_j), for all $k \in In_T$. The constraints ensure that a valid fusion structure exists to achieve the desired reduced order n' of T.

Producer constraints. First, we consider the outermost $n - l$ indices in the loop order associated with the producer C_i and ensure that they are all indices of the result reference. Specifically, for each s such that $0 \leq s < n - l$ and for each $k \in In_T$, we create terms of the form $lp_{i,k} = s$ and introduce an OR constraint for these terms (illustrated by an example below). This guarantees that the loop at position s in the loop structure surrounding the producer statement is iterating over one of the indices that appear in the result reference. The combination of these constraints for all pairs of s and k ensures that the outermost $n - l$ loops for C_i are all indices of its result tensor reference.

Example. Consider reference X_{ijqr} from the running example. This reference is produced by $C_0 : X_{ijqr} = A_{ipq} \times B_{jpr}$ and consumed by $C_1 : Y_{ijkr} = X_{ijqr} \times C_{kqr}$. We have $In_X = \{i, j, q, r\}$. The producer constraints will involve variables $lp_{0,i}$, $lp_{0,j}$, $lp_{0,q}$, and $lp_{0,r}$. Suppose $l = 2$. We would like the outermost $n - l = 4 - 2$ indices in the loop order for C_0 to be indices that access this reference. Together with the remaining constraints described shortly, this would allow those two indices to be removed from the reference after fusion. As a result, the order of X can be reduced from 4 to 2. Two constraints are formulated. First,

$$lp_{0,i} = 0 \vee lp_{0,j} = 0 \vee lp_{0,q} = 0 \vee lp_{0,r} = 0$$

ensures that the outermost loop surrounding the producer is indexed by one of i, j, q, or r. Similarly,

$$lp_{0,i} = 1 \vee lp_{0,j} = 1 \vee lp_{0,q} = 1 \vee lp_{0,r} = 1$$

guarantees that the second-outermost loop is also indexed by one of the indices of X_{ijqr}. For the fused code shown in Figure 5, we have $lp_{0,r} = 0$ (i.e., the outermost loop for C_0 is r) and $lp_{0,j} = 1$ (i.e., the second-outermost loop is j). Thus, in the fused code, the reference to X will only contain the remaining indices i and q, as shown by `X[q,i]` in Figure 5.

Consumer constraints. Next, we create constraints for the consumer contraction C_j: the sequence of its outermost $n - l$ loops must match the sequence of the outermost $n - l$ loops for the producer

C_i. This ensures that the same sequence of $n - l$ loops surrounds both the producer and the consumer, which is required for fusion that reduces the order of the intermediate from n to n' such that $n' \leq n - (n - l) = l$. (In case the constraint solver produces a solution for which more than $n - l$ outermost loops can be fused, we can have $n' < l$.) The constraints for C_j include, for each s such that $0 \leq s < n - l$ and for each $k \in In_T$, a constraint of the form $(lp_{i,k} = s) \Rightarrow (lp_{j,k} = s)$.

Example. For X_{ijqr} and its consumer C_1, we include constraints connecting $lp_{0,k}$ and $lp_{1,k}$ for each $k \in \{i, j, q, r\}$ for $s = 0$ (i.e., the outermost loop) and $s = 1$ (i.e., the second-outermost loop).

Statements between producer and consumer. Finally, we have to consider all assignments that appear between the producer C_i and the consumer C_j in the topological sort order defined by constraint variables ap_i described earlier. For any such assignment, the sequence of the outermost $n - l$ loops that surround it must match the ones for C_i and C_j. This is needed in order to have a valid fusion structure. The corresponding constraints are of the following form, for each contraction C_r with $r \neq i$ and $r \neq j$, each s with $0 \leq s < n - l$, and each $k \in In_T$:

$$(ap_i < ap_r < ap_j) \Rightarrow ((lp_{i,k} = s) \Rightarrow (lp_{r,k} = s)).$$

4 Code Generation

This section details the process of code generation from the constraint system solution. We describe how to use this solution to generate *concrete index notation*, an IR used by the TACO compiler. This IR is then used by TACO to generate the final C code implementation for the contraction tree.

4.1 Concrete Index Notation

As discussed in Section 2, TACO [19] is a state-of-the-art code generator for sparse tensor computations. While TACO does not address the questions that our work investigates (choice of linear ordering of tensor contractions from a binary contraction tree, selection of fusion structures, and tensor layouts), it does provide code generation functionality for efficient implementations of CSF tensor representations and iteration space traversals. We use concrete index notation [18], the TACO IR that captures a computation over sparse tensors through a set of computation constructs. The two constructs relevant to our work are `forall` and `where`. A `forall` construct denotes an iteration over some index. A `where(C,P)` construct denotes a producer–consumer relationship. Here `C` represents a computation that consumes a tensor being produced by computation `P`. This construct allows the use of dense workspaces [18]; as discussed in Section 2, this is an important optimization in TACO. As an illustration, the concrete index notation we generate from the constraint solution for the running example has the following form:

```
forall(r, forall(j,
where(forall(k, forall(i, R(j, k, i) = Y(k, i) * D(r, j, k))),
      where(forall(q, forall(k, forall(i, Y(k, i) = X(q, i) * C(r, q, k)))),
      forall(p, forall(q, forall(i, X(q, i) = A(p, q, i) * B(r, j, p))))))))
```

4.2 Generating Concrete Index Notation

The constraint solver's output can be abstracted as a sequence of pairs $\langle A, \pi \rangle$, where A is an assignment for a binary contraction and π is a permutation of the indices appearing in the assignment. The permutation is defined by the values of constraint variables $lp_{i,k}$ described earlier and denotes the order of surrounding loops for A. The indices in a reference to a tensor T in A are ordered based on the values of variables $dp_{T,j}$; thus, they are consistent with the order of indices in π. The order in the sequence of pairs is defined by the values of variables ap_i and represents a topological sort order of the contraction tree. For the example discussed in the previous section, the sequence is

```
<X[r,j,q,i] = A[p,q,i] * B[r,j,p],     [r,j,p,q,i]>
<Y[r,j,k,i] = X[r,j,q,i] * C[r,q,k],   [r,j,q,k,i]>
<R[j,k,i] = Y[r,j,k,i] * D[r,j,k],     [r,j,k,i]>
```

Algorithm 1 describes the creation of the TACO IR from such an input. Function `generate` is initially invoked with the entire sequence of pairs $\langle A, \pi \rangle$ based on the constraint system's solution. At each level of recursion, the function processes a sequence S of such pairs. There are two stages of processing. In the first stage (lines 3–12), a sequence L of assignments and indices is constructed. One can think of the elements of L as representing eventual assignments and loops that will be introduced in the TACO IR. For example, an index i in L will eventually lead to the creation of `forall(i,...)`. Similarly, an assignment in L will produce an equivalent assignment in the TACO IR.

ALGORITHM 1: TACO IR Generation

```
   function generate(S):
       input: sequence S of pairs ⟨A, π⟩; A is an assignment and π is a permutation of A's indices
       output: concrete index notation for S
1      L ← empty list // L is a sequence of indices and/or assignments
2      M ← empty map // M is a map from an index to a sequence of ⟨A, π⟩
3      for k ← 0 to |S| − 1 do
4          ⟨A, π⟩ ← S_k
5          if π is empty then
6              L.append(A)
7          else
8              i ← π.first() // i is the index of the outermost loop for A at this level
9              if i ≠ L.last() then
                   // i does not match the last element of L and should be added to L
10                 L.append(i)
11                 M.put(i, empty list)
12             M.get(i).append(⟨A, π⟩)
13     if L.length() == 1 then
14         if L.first() is an assignment A then return A
15         if L.first() is an index i then
               // single index i in L; create a 'forall' construct for i
16             return forall( i , generate (remove (i, M.get(i))) )
17     else
           // several indices and/or assignments in L; create a 'where' construct
18         return where( generate (M.get(L.last())) , generate (S.truncate(M.get(L.last()))) )
```

During this first stage, for each element $\langle A, \pi \rangle$ of S, in order, we need to decide whether the loop structure encoded by π can be fused with the loop structure of the previous element of S, at this level of loop nesting. For example, the sequence shown above contains permutation `[r,j,p,q,i]` in the first pair of S, followed by `[r,j,q,k,i]` in the second pair. The processing of the first pair will add index `r` to L. In the processing of the second pair, the outermost index `r` matches the current last element of L, and thus `r` is a common loop for both assignments. The processing of the third pair considers permutation `[r,j,k,i]`, whose outermost index again matches the last element of L. Thus, at the end of the stage, L contains one element: the index `r`. In a more general case, a combination of indices and assignments could be added to L. For example, if the input sequence is `<A0,[i]>`, `<A1,[]>`, L contains two elements—`i` followed by `A1`—which eventually leads to the creation of `where(A1,forall(i,A0))`, as described shortly.

As part of this process, for each index in L the algorithm records the sub-sequence of relevant pairs from S. This information is stored in map M, with keys being the indices that are recorded in L. For the running example, `r` is mapped in M to the sequence of all three input pairs. This list of pairs is then used in the second stage of processing to generate a construct of the form `forall(r,...)`.

The second stage (lines 13–18) considers three cases. If L contains a single assignment, this assignment simply becomes the result of IR generation (line 14). If L contains a single index i, this index can be used to create a `forall(i,...)` construct that surrounds all pairs recorded in $M.get(i)$. This creation is shown at line 16. The pairs in $M.get(i)$ are first processed by a helper function `remove` and then used to recursively generate the body of the `forall`. The helper function, which is not shown in the algorithm, plays two roles. Both are illustrated by the modified pairs below, which are obtained by calling `remove(r,`$M.get$`(r))`:

```
<X[j,q,i] = A[p,q,i] * B[r,j,p],   [j,p,q,i]>
<Y[j,k,i] = X[j,q,i] * C[r,q,k],   [j,q,k,i]>
<R[j,k,i] = Y[j,k,i] * D[r,j,k],   [j,k,i]>
```

First, `remove` eliminates `r` from the start of all permutations π. This reflects the fact that a `forall(r,...)` is created at line 16. Second, the function removes `r` from all intermediate tensor references for which both the producer and the consumer are in $M.get$(`r`). For example, `X[r,j,q,i]` appears in the first pair (the producer) and in the second pair (the consumer). Both are surrounded by the common loop `r`, which means that `X` can be reduced from order-4 to order-3, and thus the reference is rewritten as `X[j,q,i]`. A similar change is applied to `Y[r,j,k,i]`.

At the next level of recursion, this sequence becomes the input to `generate`. During that processing, L contains only index `j` and `remove(j,`$M.get$`(j))` is called to obtain the modified sequence:

```
<X[q,i] = A[p,q,i] * B[r,j,p],   [p,q,i]>
<Y[k,i] = X[q,i] * C[r,q,k],     [q,k,i]>
<R[j,k,i] = Y[k,i] * D[r,j,k],   [k,i]>
```

Then `generate` is called on this sequence. At that level of recursion, L contains three indices: `p`, `q`, and `k`. This illustrates the third case in the processing of L. Line 18 shows the creation of a `where` construct for this case. Since `k` is the last element of L, the first operand of `where` is the IR generated for the sub-sequence corresponding to `k`, which here contains a single pair `<R[j,k,i] = Y[k,i] * D[r,j,k], [k,i]>`.

Recall that this first operand of `where` corresponds to a consumer of a tensor—in this case, tensor `Y`. The producer of `Y` appears in the second operand of `where`, which is generated from the first two pairs from the original sequence:

```
<X[q,i] = A[p,q,i] * B[r,j,p],   [p,q,i]>
<Y[k,i] = X[q,i] * C[r,q,k],     [q,k,i]>
```

At line 18, $S.truncate$ denotes an operation to produce this desired prefix of S by excluding the sub-sequence defined by $M.get(L.last())$. The IR generated from this prefix itself contains a nested `where` construct, which captures a producer–consumer computation for `X`. At the end of processing, the resulting overall structure has the form

```
forall(r, forall(j, where(forall(k, forall(i, A2)),
                          where(forall(q, forall(k, forall(i, A1))),
                                forall(p, forall(q, forall(i, A0)))))))
```

Table 2. Z3 Constraints: Dimensionality of Intermediates, Number of Constraint Variables, Number of Constraints, Solver Time

Tensor network	DimBound	ConsVars	Constraints	SolverTime (s)
3-Index unrestricted	1	19	38	0.013
3-Index restricted	1	27	57	0.015
4-Index	1	22	46	0.011
MTTKRP	1	17	40	0.01
TTMc	1	19	38	0.01

5 Experimental Evaluation

Benchmarks. We evaluate the performance of CoNST-generated code on several sparse tensor networks. Section 5.1 presents a case study of sparse tensor computations arising from recent developments with linear-scaling methods in quantum chemistry [33]. Three tensor networks are used: 3-index integral unrestricted, 3-index integral restricted, and 4-index integral; details on these networks are provided in Section 5.1. Section 5.2 evaluates performance on the **Matricized Tensor Times Khatri-Rao Product (MTTKRP)** computation [20]. Section 5.3 presents performance on the **Tensor Times Matrix chain (TTMc)** expression that is the performance bottleneck for the Tucker decomposition algorithm [20].

Constraint systems. For each of these benchmarks, Table 2 provides details on the Z3 constraint system that was solved to generate the code for our performance evaluations. Column "DimBound" shows the upper bound l on the dimensionality of intermediate tensors; recall from Section 3 that this parameter l is used when generating the constraints.[3] Column "ConsVars" shows the number of constraint variables, while column "Constraints" shows the number of constraints. The last column "SolverTime" shows the execution time of Z3. As can be seen from these measurements, the systems are relatively small and their solutions can be computed very quickly. The follow-up steps of generating the TACO IR and then generating executable code with TACO are also quick, and together take about 0.1 second.

Recall that fusion allows for reduction in the dimensionality (and thus memory usage) of intermediate tensors. As shown in column "DimBound" in Table 2, the solver can be used to identify fusion structures with low-dimensional intermediates. Comparing with the memory usage in an unfused version (configuration TACO-Unfused, described shortly), we observed that without fusion the memory usage for intermediates in our benchmarks is typically a few megabytes, while the CoNST-generated fusion reduces this memory usage to a few kilobytes.

Performance evaluation. All experiments were conducted on an AMD Ryzen Threadripper 3990X 64-core processor with 128 GB RAM. Optimization flags `-O3 -fast-math` were used to compile the C code, with the GCC 9.4 compiler. Reported performance results are for single-thread execution. Effective parallelization of the code is a topic for future work. A key challenge is that of achieving effective load balancing across threads because of the significant variance in the work for different iterations of outer-most parallel loops, due to highly variable index-dependent sparsity of inner nested loops. In all experiments and for all evaluated tools, the input tensors are in COO format on disk. The time to read the tensors from disk and to represent them in the CSF formats required by the tools is not included in the measurements.

We compare CoNST against three state-of-the-art sparse tensor compilers and libraries:

[3] We have also tested the approach with synthetic examples where $l > 1$ is needed to achieve a feasible constraint solution.

TACO:[4] As discussed in detail earlier, CoNST uses TACO for generation of C code after co-optimization for tensor layout choice, schedule for the contractions, loop fusion, and mode reduction of intermediate tensors. We compare the performance of CoNST-generated code with that achieved by direct use of TACO. This was done in two ways: (1) direct N-ary contraction code was generated by TACO, where a single multi-term tensor product expression was provided as input with the same mode order for tensors produced by CoNST's constraint solver (described in Section 3), and (2) TACO was used to generate code for an unfused sequence of binary contractions, in which case results are reported for the best-performing mode order for tensors.

SparseLNR:[5] SparseLNR takes a multi-term tensor product expression and generates fused code for it by transforming it internally to a sequence of binary contractions. We evaluated its performance by providing the same multi-term tensor expression used for comparison with TACO.

Sparta:[6] We used Sparta to compute the sequence of binary tensor contractions produced by CoNST. However, Sparta's kernel implementation internally requires that the contraction index be at the inner-most mode for one input tensor and at the outer-most mode for the other input tensor. If the provided input tensors do not satisfy this condition, explicit tensor transposition is performed by Sparta before performing the sparse tensor contraction. Since the tensor layout generated by CoNST might not conform to Sparta's constraints, we instead performed an exhaustive study that evaluated all combinations of distinct tensor layout orders that would not need additional transpositions for Sparta. We report the lowest execution time among all evaluated configurations.

5.1 Computing Sparse Integral Tensors for DLPNO Methods in Quantum Chemistry

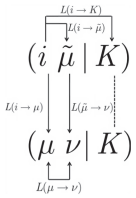
Recent developments in predictive-quality quantum chemistry have sought to reduce their computational complexity from a high-order polynomial in the number of electrons N (e.g., $O(N^7)$ and higher for predictive-quality methods like coupled-cluster [4]) to linear in N, by exploiting various types of sparsity of electronic wave functions and the relevant quantum mechanical operators [35].

The few efficient practical realizations of **Domain-based Local Pair Natural Orbital (DLPNO)** and other similar methods, e.g., the Orca package [28], have developed custom implementations of sparse tensor algebra, without any utilization of generic infrastructure for sparse tensor computations. Below we present a case study that demonstrates the potential for using CoNST to automatically generate code that can address the kinds of sparsity constraints that arise in the implementation of DLPNO and similar sparse formulations in quantum chemistry.

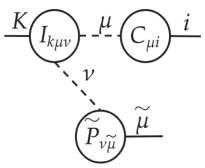
A key step in the DLPNO methods is the evaluation of matrix elements (integrals) of the electron repulsion operator that was first formulated in a linear-scaling fashion by Pinski et al. [33]. The first stage of the DLPNO integral evaluation involves a multi-term tensor product of three sparse tensors; Figure 6(b) shows a sparse tensor network corresponding to the expression: $E_{Ki\tilde{\mu}} = I_{K\mu\nu} \times C_{\mu i} \times \tilde{P}_{\nu\tilde{\mu}}$. The indices of the tensors correspond to four pertinent spaces, ordered from least to most numerous: (1) localized *molecular orbitals* (indexed in the code by i), (2) *atomic orbitals* (indexed by μ and ν), (3) *projected atomic orbitals* [34] (indexed by $\tilde{\mu}$), and (4) density fitting atomic orbitals (indexed by K).

The sparse structure of tensors as well as the ranges of loops in the code are governed by various sparsity relationships or *sparse maps* between pairs of index spaces, as illustrated in Figure 6(a) (reproduced from Pinski et al. [33]). This enables a reduction of the number of executed operations, and only a subset of all elements of this tensor network are evaluated. Figure 6(c) shows a four-term

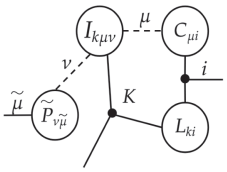
[4]TACO code: https://github.com/tensor-compiler/taco
[5]SparseLNR code: https://github.com/adhithadias/SparseLNR
[6]Sparta code: https://github.com/pnnl/HiParTI/tree/sparta



(a) Sparse maps involved in the computation of DLPNO integrals (reproduced from [33]).



(b) Sparse tensor network for unrestricted evaluation of DLPNO integrals



(c) Sparse tensor network for 3-centered integral, with sparse tensor L_{Ki} to impose additional sparsity in result tensor

Fig. 6. Sparse integral tensor case study.

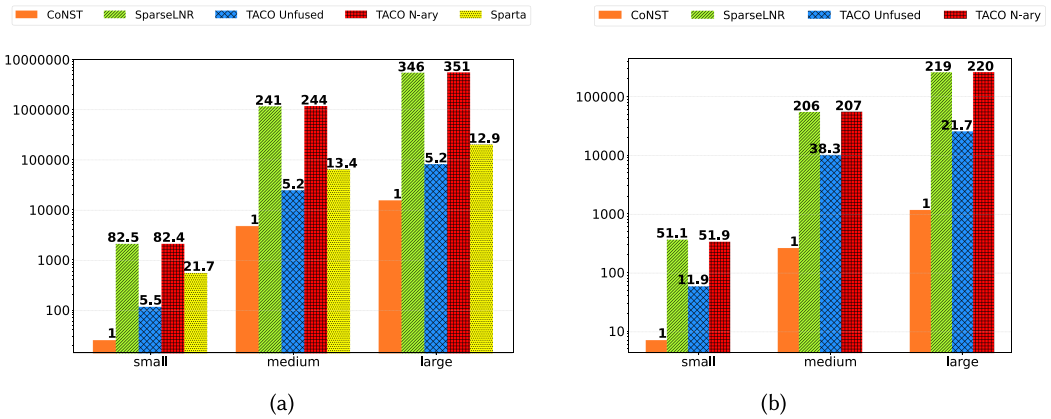


Fig. 7. Execution time (ms) for evaluation of 3-index integrals (lower is better; Y-axis is in logarithmic scale) using (a) unrestricted and (b) restricted tensor networks, respectively. Numbers above the bars represent slowdown of other schemes relative to CoNST.

sparse tensor network where an additional 0/1 sparse matrix L_{Ki} has been added to the base tensor network in Figure 6(b), corresponding to the known sparse map $L(K \rightarrow i)$. This can equivalently be expressed as a multi-term tensor product expression: $E_{Ki\tilde{\mu}} = I_{K\mu\nu} \times C_{\mu i} \times \tilde{P}_{\nu\tilde{\mu}} \times L_{Ki}$. The inclusion of such sparse maps as additional nodes in the base tensor network has the same beneficial effect of reducing computations as the manually implemented restriction in Orca [28]. In our experimental evaluation, we evaluate both forms of the sparse tensor networks in Figure 6, representing the *unrestricted* form (Figure 6(b)) and the *restricted* form (Figure 6(c)).

We computed the DLPNO integrals for 2-D solid helium lattices with the geometry described in [23]. The "small" input used a 5×5 lattice (25 atoms) and "medium"/"large" inputs used a 10×10 lattice (100 atoms). The following orbital and density-fitting basis sets were used: 6-311G [13] and the spherical subset of def2-QZVPPD-RIFIT [14], respectively, for the "small" and "medium" inputs, and cc-pVDZ [47] and cc-pVDZ-RIFIT [46] for the "large" input. All quantum chemistry data was prepared using the Massively Parallel Quantum Chemistry package [31].

Figure 7(a) presents measurements for the transformed 3-index integral $E_{Ki\tilde{\mu}}$ in unrestricted form (Figure 6(b)). CoNST-generated code is about two orders of magnitude faster than the N-ary code generated by TACO as well as SparseLNR (for this case SparseLNR was unable to perform loop fusion and simply lowered the input to TACO). TACO-Unfused is much faster than N-ary but is still about five to six times slower than the code generated by CoNST. The best of the comprehensively

evaluated versions for Sparta is about an order of magnitude slower than CoNST's code. We note that a direct comparison with the domain-specific implementation in Orca [28] is very challenging because its implementation of DLPNO-CC fuses tensor contraction with other computation. For example, the 3-index MO integral evaluation fuses contraction with the evaluation of AO integrals, and the 4-index integral evaluation in ORCA uses pre-computed 3-index integrals stored on disk.

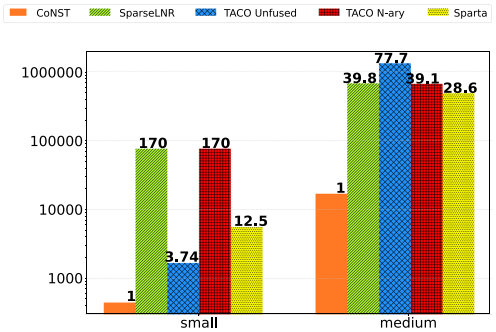


Fig. 8. Execution time (ms) for 4-index integral. Numbers above the bars represent slowdown relative to CoNST.

The performance data for evaluation of $E_{Ki\tilde{\mu}}$ using the restricted form (Figure 6(c)) is presented in Figure 7(b). Significant speedups can be seen between the execution times in Figure 7(a) and 7(b) (the Y-axis scales are different) by use of the additional tensor L_{Ki} for CoNST, SparseLNR, and TACO N-ary, with the speedup with use of CoNST being roughly the same. However, TACO-Unfused does not improve as much, causing its slowdown with respect to CoNST to get worse. No data for Sparta is presented in Figure 7(b) because of a constraint of Sparta that a tensor product must have a contraction index, which is not the case for the tensor product with L_{Ki}.

A subsequent step after formation of the 3-centered integrals is to use them to construct 4-index integrals (Equation (16) in Reference 33): $V_{ij\tilde{\mu}\tilde{\nu}} = E_{Ki\tilde{\mu}} \times E_{Kj\tilde{\nu}}$, using the 3-index input tensor E obtained via the unrestricted form.

Performance results are reported in Figure 8. CoNST again achieves significant speedup over the alternatives. For this experiment, we could not use the large dataset because of insufficient physical memory on our platform.

5.2 Sparse Tensor Network for CP Decomposition

Canonical Polyadic (CP) decomposition factorizes a sparse tensor T with n modes into a product of n 2-D matrices. For example, a 3-D tensor T_{ijk} is decomposed into three dense rank-r matrices A_{ir}, B_{jr}, and C_{kr}. The CP decomposition of a sparse tensor is generally performed using an iterative algorithm that requires n **Matricized Tensor Times Khatri-Rao Product (MTTKRP)** operations [20]. For a 3-D tensor, the three MTTKRP operations are as follows:

$$A'_{ir} = T_{ijk} \times B_{jr} \times C_{kr} \qquad B'_{jr} = T_{ijk} \times A_{ir} \times C_{kr} \qquad C'_{kr} = T_{ijk} \times A_{ir} \times B_{jr}.$$

Figure 9 shows the performance for MTTKRP operations for each of the three modes for sparse tensors from the FROSTT benchmark suite [39]. We used the same four sparse tensors (Flickr3d, Nell1, Nell2, and Vast3d) used in the experimental evaluation of SparseLNR [12]. The rank of factor matrices was set to 50. The time to perform the MTTKRP operation for the three modes varies quite significantly; this is in part due to the the highly non-uniform extents of the three modes for the tensors (as seen in Table 3). For the MTTKRP expression, SparseLNR was not able to perform its *loopFusionOverFission* transformation, so the code and performance are essentially identical to TACO N-ary. Considering CoNST, unlike with the DLPNO benchmark (Section 5.1), CoNST-generated code is not always fastest. For the first two operations, CoNST achieves a speedup between 2.0× and 4.8× over other schemes, but relative performance is lower for the third one, ranging between 0.9× and 1.0× over the best alternative. For this case, the size of the intermediate tensor is small and the binary tensor contractions are efficient without fusion, whereas the mode

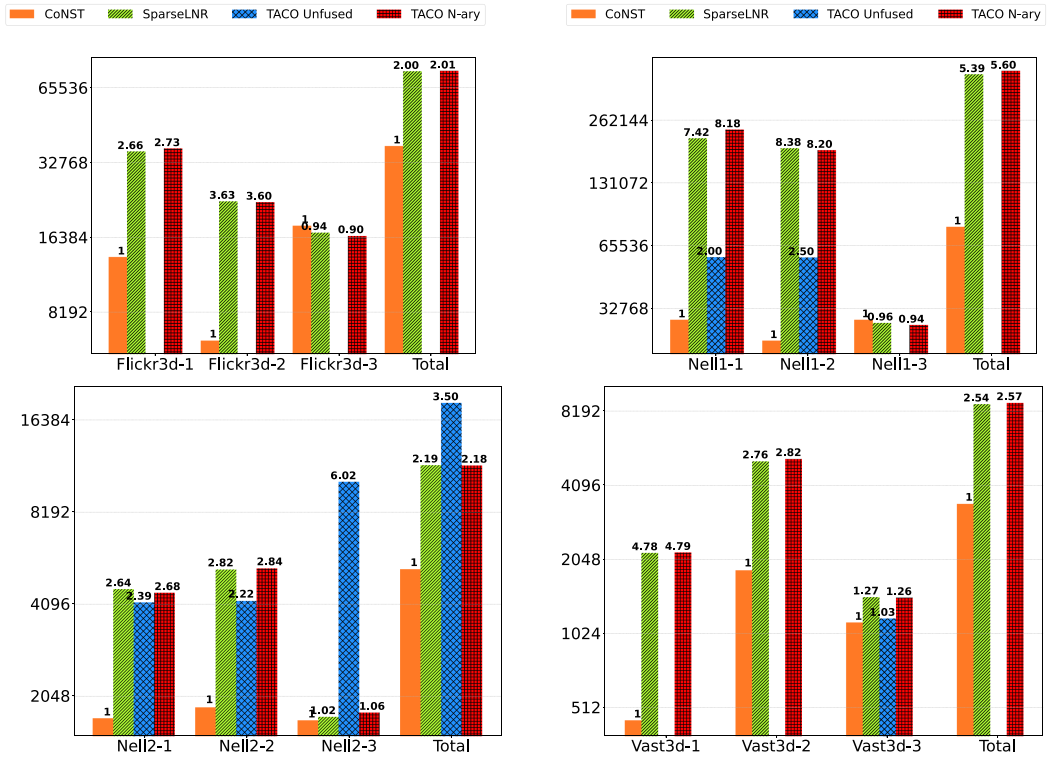


Fig. 9. Execution time (ms) for MTTKRP operations on the FROSTT tensors. Relative slowdown compared to CoNST is indicated above each bar. Missing bars mean out-of-memory failure (for TACO-Unfused).

Table 3. FROSTT Tensors and Their Shapes

Tensor	Dimensions			NNZs
flickr-3d	320K	2.82M	1.6M	112.89M
nell-2	12K	9K	288K	76.88M
nell-1	2.9M	2.14M	25.5M	143.6M
vast-2015-mc1-3d	165K	11K	2	26.02M

order that enables fusion results in code with slightly lower performance than the unfused code. However, when considering the total time for all three operations, as needed in each iteration in CP decomposition, CoNST achieves a minimum speedup of 2× over the alternatives, across the four benchmarks. Sparta times are not reported because it could not be used: it does not handle contractions with "batch" indices that occur in both input tensors and output tensors, as occurs with the second tensor contraction in the binarized sequence for each MTTKRP.

5.3 Sparse Tensor Network for Tucker Decomposition

Tucker decomposition factorizes a sparse tensor T with n modes into a product of n 2-D matrices and a dense *core* n-mode tensor. For example, a 3-D tensor T_{ijk} is decomposed into three rank-r matrices A_{ix}, B_{jy}, C_{kz}, and core tensor G_{xyz}. The computation is generally performed using the **High Order Orthogonal Iteration (HOOI)** iterative algorithm that requires n **Tensor Times**

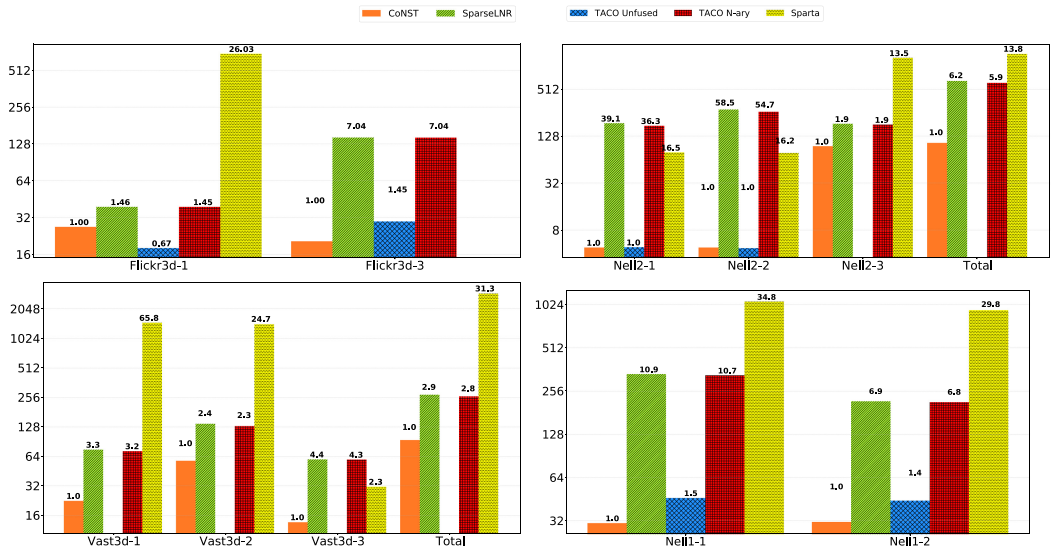


Fig. 10. Execution time (ms) for TTMc operations on the FROSTT tensors. Relative slowdown compared to CoNST is shown above the bar. Missing bars indicate out-of-memory failure.

Matrix chain (TTMc) operations [20]. For a 3-D tensor, the TTMc operations are as follows:

$$A'_{iyz} = T_{ijk} \times B_{jy} \times C_{kz} \qquad B'_{jxz} = T_{ijk} \times A_{ix} \times C_{kz} \qquad C'_{kxy} = T_{ijk} \times A_{irxr} \times B_{jryr}.$$

Figure 10 presents execution times for the alternative schemes on the four FROSTT tensors. The mode-2 contraction for Flickr3d and mode-3 contraction for Nell-1 tensor ran out of memory for all methods on 128GB RAM. TACO-Unfused and Sparta ran out of memory for a larger set of runs because they form high-dimensional sparse intermediates in memory. The rank of decomposition was 16 for Nell-1 and Flickr-3d tensors, and 50 for Vast-3d and Nell-2 tensors. For the TTMc operation, SparseLNR is not able to perform its *loopFusionOverFission* transformation, so that performance is identical to TACO N-ary. Sparta runs a flattened matrix-times-matrix operation for a general tensor contraction and uses a hashmap to accumulate rows of the result. Since the matrix being multiplied is dense, the hashmap simply adds an overhead. Overall, CoNST generates code that achieves significant speedups over the compared alternatives.

6 Related Work

A comparison between CoNST and the three most related prior efforts was presented in Section 2 and summarized in Table 1. As shown in the previous section, significant performance improvements can be achieved by the code generated through CoNST's integrated treatment of contraction/loop/mode order for fused execution of general contraction trees, compared to (1) code directly generated by TACO [19], (2) fused loop code generated by SparseLNR [12], and (3) calls to the Sparta library [25] for sparse tensor contractions. A variety of sparse formats are possible in TACO through format abstractions [9]; our work focuses on the popular CSF representation. We note that very recent developments [49] have advanced the TACO compiler to allow some forms of sparse workspaces for intermediate tensors; our work was completed before this feature was available and only uses dense workspaces [18] previously supported by TACO.

A number of efforts have addressed loop-level optimization of dense tensor contractions for CPUs and GPUs [1, 17, 22, 26, 29, 36, 37, 43]. However, none of them can be directly adapted for

optimizing sparse tensor contractions because of the need to use a compact data representation for sparse tensors. Cociorva et al. [10] addressed loop fusion in the context of optimizing dense tensor networks. Similar to CoNST, the reduction of the order of intermediate tensors is addressed. However, the main focus of that work was to identify more aggressively fused configurations that further reduced the space for intermediate tensors at the price of redundant computation. CoNST does not introduce redundant operations in its optimization and does not incorporate any such space–time tradeoff considerations.

Sparse fusion [7] is an inspector–executor strategy for iteration composition for fused execution of two sparse kernels. This approach optimizes kernels with loop-carried dependencies using runtime techniques. Tensor mode layout and its interactions with iteration order and mode reduction of intermediate sparse tensors are not considered by this existing work. In contrast, our work considers these factors in compile-time code generation for a general contraction tree.

The sparse polyhedral framework [44] defines inspector–executor techniques for optimization of sparse computations, e.g., through runtime iteration/data reordering. It has been applied to individual tensor contractions [50] where iteration code is derived using polyhedral scanning. This approach does not consider fusion or reordering of loops/modes. In contrast to inspector–executor approaches, which often involve non-trivial "inspector" overhead in analyzing the sparsity pattern for each execution instance, CoNST uses a purely static compile-time approach. It thus avoids such overhead and the generated code can be used for different input tensors without any instance-specific runtime analysis.

SparseTIR [48] is an approach to represent sparse tensors in composable formats and to enable program transformations in a composable manner. The sparse compilation support in the MLIR infrastructure [5] enables integration of sparse tensors and computations with other elements of MLIR, as well as TACO-like code generation. SpTTN-Cyclops [16] is an extension of the **Cyclops Tensor Framework (CTF)** [42] to optimize a sub-class of sparse tensor networks. In contrast to CoNST, which can handle arbitrary sparse tensor networks, SpTTN-Cyclops only targets a product of a single sparse tensor with a network of several dense tensors. Indexed Streams [21] develops a formal operational model and intermediate representation for fused execution of tensor contractions, using both sparse tensor algebra and relational algebra, along with a compiler to generate code. Tian et al. [45] introduce a DSL to support dense and sparse tensor algebra algorithms and sparse tensor storage formats in the COMET compiler [27], which generates code for a given tensor expression. Zhou et al. [51] propose a set of techniques to optimize tensor networks including use of loop fusion. Using manual implementation of the proposed techniques, performance improvement is demonstrated over code generated by TACO [19] on a set of benchmarks. Finch [2, 3] is a recently developed framework that supports efficient code generation for computations on sparse matrices/tensors that exhibit structured sparsity. None of these efforts address the coupled automated optimization of tensor layout, contraction schedule, and mode reduction for intermediates in fused code being performed by CoNST.

7 Conclusions

Effective fused code generation for sparse tensor networks depends on inter-related factors: schedule of binary contractions, permutation of nested loops, and layout order of tensor modes. We demonstrate that an integrated constraint-based formulation can capture these factors and can produce fused loop structures for efficient execution. Our experimental evaluation confirms that this approach advances the state of the art in achieving high performance for sparse tensor networks. An important next step is to apply these techniques for sparse tensor algebras needed by computational scientists (e.g., in quantum chemistry [6]).

Acknowledgments

We thank the ACM TACO reviewers for their valuable feedback.

References

[1] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobrev, Jack Dongarra, Christopher Earl, Joël Falcou, Azzam Haidar, Ian Karlin, Tzanio Kolev, Ian Masliah, and Stanimire Tomov. 2016. High-performance tensor contractions for GPUs. *Procedia Computer Science* 80 (2016), 108–118.

[2] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2024. Finch: Sparse and structured array programming with control flow. *arXiv preprint arXiv:2404.16730* (2024).

[3] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A language for structured coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. 41–54.

[4] Rodney J. Bartlett and Monika Musiał. 2007. Coupled-cluster theory in quantum chemistry. *Reviews of Modern Physics* 79, 1 (2007), 291.

[5] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler support for sparse tensor computations in MLIR. *ACM Transactions on Architecture and Code Optimization* 19, 4, Article 50 (2022), 25 pages.

[6] Justus A. Calvin, Chong Peng, Varun Rishi, Ashutosh Kumar, and Edward F. Valeev. 2021. Many-body quantum chemistry on massively parallel computers. *Chemical Reviews* 121, 3 (Feb. 2021), 1203–1231. DOI: https://doi.org/10.1021/acs.chemrev.0c00006

[7] Kazem Cheshmi, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2023. Runtime composition of iterations for fusing loop-carried sparse dependence. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Article 89, 15 pages.

[8] Lam Chi-Chung, P. Sadayappan, and Rephael Wenger. 1997. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters* 7, 2 (1997), 157–168.

[9] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages.

[10] Daniel Cociorva, Gerald Baumgartner, Chi-Chung Lam, P. Sadayappan, J. Ramanujam, Marcel Nooijen, David E. Bernholdt, and Robert Harrison. 2002. Space-time trade-off optimization for a class of electronic structure calculations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. 177–186.

[11] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

[12] 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*. 1–14.

[13] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, G. Scalmani, V. Barone, B. Mennucci, G. A. Petersson, H. Nakatsuji, M. Caricato, X. Li, H. P. Hratchian, A. F. Izmaylov, J. Bloino, G. Zheng, J. L. Sonnenberg, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, T. Vreven, J. A. Montgomery, Jr., J. E. Peralta, F. Ogliaro, M. Bearpark, J. J. Heyd, E. Brothers, K. N. Kudin, V. N. Staroverov, R. Kobayashi, J. Normand, K. Raghavachari, A. Rendell, J. C. Burant, S. S. Iyengar, J. Tomasi, M. Cossi, N. Rega, J. M. Millam, M. Klene, J. E. Knox, J. B. Cross, V. Bakken, C. Adamo, J. Jaramillo, R. Gomperts, R. E. Stratmann, O. Yazyev, A. J. Austin, R. Cammi, C. Pomelli, J. W. Ochterski, R. L. Martin, K. Morokuma, V. G. Zakrzewski, G. A. Voth, P. Salvador, J. J. Dannenberg, S. Dapprich, A. D. Daniels, Ö. Farkas, J. B. Foresman, J. V. Ortiz, J. Cioslowski, and D. J. Fox. [n. d.]. Gaussian 09 Revision E.01. Gaussian Inc., Wallingford, CT.

[14] Christof Haettig. 2005. Optimization of auxiliary basis sets for RI-MP2 and RI-CC2 calculations: Core-valence and quintuple-ζ basis sets for H to Ar and QZVPP basis sets for Li to Kr. *Physical Chemistry Chemical Physics: PCCP* 7 (Jan. 2005), 59–66. DOI: https://doi.org/10.1039/B415208E

[15] Albert Hartono, Alexander Sibiryakov, Marcel Nooijen, Gerald Baumgartner, David E. Bernholdt, So Hirata, Chi-Chung Lam, Russell M. Pitzer, J. Ramanujam, and P. Sadayappan. 2005. Automated operation minimization of tensor contraction expressions in electronic structure calculations. In *Proceedings of the 5th International Conference on Computational Science (ICCS'05), Part I 5*. Springer, 155–164.

[16] Raghavendra Kanakagiri and Edgar Solomonik. 2023. Minimum cost loop nests for contraction of a sparse tensor with a tensor network. *arXiv preprint arXiv:2307.05740* (2023).

[17] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2019. A code generator for high-performance tensor contractions on GPUs. In *International Symposium on Code Generation and Optimization (CGO'19)*. 85–95.

[18] 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'19)*. 180–192.

[19] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

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

[21] Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A formal intermediate representation for fused contraction programs. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1169–1193.

[22] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and Ponnuswamy Sadayappan. 2019. Analytical cache modeling and tilesize optimization for tensor contractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[23] Weitong Lin, Yiran Li, Sytze Graaf, Gang Wang, Junhao Lin, Hui Zhang, Shijun Zhao, Da Chen, Shaofei Liu, Jun Fan, B. J. Kooi, Tao Yang, Chin-Hua Yang, Chain Liu, and Ji-jung Kai. 2022. Creating two-dimensional solid helium via diamond lattice confinement. *Nature Communications* 13 (Oct. 2022), 5990. DOI: https://doi.org/10.1038/s41467-022-33601-5

[24] Jiawen Liu, Dong Li, Roberto Gioiosa, and Jiajia Li. 2021. Athena: High-performance sparse tensor contraction sequence on heterogeneous memory. In *Proceedings of the ACM International Conference on Supercomputing*. 190–202.

[25] 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*. 318–333.

[26] Devin A. Matthews. 2018. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing* 40, 1 (2018), C1–C24.

[27] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2020. COMET: A domain-specific compilation of high-performance computational chemistry. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 87–103.

[28] Frank Neese, Frank Wennmohs, Ute Becker, and Christoph Riplinger. 2020. The ORCA quantum chemistry program package. *Journal of Chemical Physics* 152, 22 (2020), 224108.

[29] Thomas Nelson, Axel Rivera, Prasanna Balaprakash, Mary Hall, Paul D. Hovland, Elizabeth Jessup, and Boyana Norris. 2015. Generating efficient tensor contractions for GPUs. In *2015 44th International Conference on Parallel Processing*. IEEE, 969–978.

[30] Nvidia. 2020. cuTENSOR: A High-performance CUDA Library for Tensor Primitives. https://docs.nvidia.com/cuda/cutensor/index.html

[31] Chong Peng, Cannada A. Lewis, Xiao Wang, Marjory C. Clement, Karl Pierce, Varun Rishi, Fabijan Pavošević, Samuel Slattery, Jinmei Zhang, Nakul Teke, Ashutosh Kumar, Conner Masteran, Andrey Asadchev, Justus A. Calvin, and Edward F. Valeev. 2020. Massively parallel quantum chemistry: A high-performance research platform for electronic structure. *Journal of Chemical Physics* 153, 4 (Jul. 2020), 044120. DOI: https://doi.org/10.1063/5.0005889 arXiv:https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0005889/16709494/044120_1_online.pdf

[32] Robert N. C. Pfeifer, Jutho Haegeman, and Frank Verstraete. 2014. Faster identification of optimal contraction sequences for tensor networks. *Physical Review E* 90, 3 (2014), 033315.

[33] Peter Pinski, Christoph Riplinger, Edward F. Valeev, and Frank Neese. 2015. Sparse maps—A systematic infrastructure for reduced-scaling electronic structure methods. I. An efficient and simple linear scaling local MP2 method that uses an intermediate basis of pair natural orbitals. *Journal of Chemical Physics* 143, 3 (2015), 034108.

[34] Peter Pulay. 1983. Localizability of dynamic electron correlation. *Chemical Physics Letters* 100, 2 (1983), 151–154.

[35] Christoph Riplinger, Peter Pinski, Ute Becker, Edward F. Valeev, and Frank Neese. 2016. Sparse maps—A systematic infrastructure for reduced-scaling electronic structure methods. II. Linear scaling domain based pair natural orbital coupled cluster theory. *Journal of Chemical Physics* 144, 2 (2016), 024109.

[36] Swarup Kumar Sahoo, Sriram Krishnamoorthy, Rajkiran Panuganti, and P. Sadayappan. 2005. Integrated loop optimizations for data locality enhancement of tensor contraction expressions. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC'05)*. IEEE, 13–13.

[37] Yang Shi, Uma Naresh Niranjan, Animashree Anandkumar, and Cris Cecka. 2016. Tensor contractions with extended BLAS kernels on CPU and GPU. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC'16)*. IEEE, 193–202.

[38] Daniel G. A. Smith and Johnnie Gray. 2018. opt_einsum—A Python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software* 3, 26 (2018), 753.

[39] 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*. http://frostt.io/

[40] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. Article 5, 7 pages.

[41] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 61–70.

[42] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 813–824.

[43] Paul Springer and Paolo Bientinesi. 2018. Design of a high-performance GEMM-like tensor–tensor multiplication. *ACM Transactions on Mathematical Software (TOMS)* 44, 3 (2018), 1–29.

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

[45] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A high-performance sparse tensor algebra compiler in multi-level IR. *arXiv preprint arXiv:2102.05187* (2021).

[46] Florian Weigend, Andreas Köhn, and Christof Hättig. 2002. Efficient use of the correlation consistent basis sets in resolution of the identity MP2 calculations. *Journal of Chemical Physics* 116, 8 (Feb. 2002), 3175–3183. DOI: https://doi.org/10.1063/1.1445115. arXiv:https://pubs.aip.org/aip/jcp/article-pdf/116/8/3175/10841034/3175_1_online.pdf

[47] David E. Woon and Thom H. Dunning Jr. 1994. Gaussian basis sets for use in correlated molecular calculations. IV. Calculation of static electrical response properties. *Journal of Chemical Physics* 100, 4 (Feb. 1994), 2975–2988. DOI: https://doi.org/10.1063/1.466439. arXiv:https://pubs.aip.org/aip/jcp/article-pdf/100/4/2975/10771441/2975_1_online.pdf

[48] 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*. 660–678.

[49] Genghan Zhang, Olivia Hsu, and Fredrik Kjolstad. 2024. Compilation of modular and general sparse workspaces. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1213–1238.

[50] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral specification and code generation of sparse tensor contraction with co-iteration. *ACM Transactions on Architecture and Code Optimization* 20, 1 (2022), 1–26.

[51] Tong Zhou, Ruiqin Tian, Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, and Vivek Sarkar. 2022. ReACT: Redundancy-aware code generation for tensor expressions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 1–13.

Received 25 June 2024; revised 25 June 2024; accepted 30 July 2024