Minimum Cost Loop Nests for Contraction of a Sparse Tensor with a Tensor Network

Raghavendra Kanakagiri
Department of Computer Science
University of Illinois Urbana-Champaign
Urbana, IL, USA
Indian Institute of Technology Tirupati
Tirupati, AP, India
raghavendra@iittp.ac.in

Edgar Solomonik
Department of Computer Science
University of Illinois Urbana-Champaign
Urbana, IL, USA
solomon2@illinois.edu

ABSTRACT

Sparse tensor decomposition and completion are common in numerous applications, ranging from machine learning to computational quantum chemistry. Typically, the main bottleneck in optimization of these models are contractions of a single large sparse tensor with a network of several dense matrices or tensors (SpTTN). Prior works on high-performance tensor decomposition and completion have focused on performance and scalability optimizations for specific SpTTN kernels. We present algorithms and a runtime system for identifying and executing the most efficient loop nest for any SpTTN kernel. We consider both enumeration of such loop nests for autotuning and efficient algorithms for finding the lowest cost loop nest for simpler metrics, such as buffer size or cache miss models. Our runtime system identifies the best choice of loop nest without user guidance, and also provides a distributed-memory parallelization of SpTTN kernels. We evaluate our framework using both real-world and synthetic tensors. Our results demonstrate that our approach outperforms available generalized state-of-the-art libraries and matches the performance of specialized codes.

CCS CONCEPTS

• **Theory of computation** → **Dynamic programming**; **Massively parallel algorithms**; • **Software and its engineering** → **Runtime environments**.

KEYWORDS

Sparse Tensor Algebra, Tensor Contraction, Tensor Decomposition and Completion

1 INTRODUCTION

Tensors provide a mathematical representation for multi-dimensional arrays, enabling basic operations such as contraction (composition) and decomposition of tensors. Tensor contraction and decomposition are used in many methods for modeling quantum systems [22, 29, 42, 45] and to construct models of data in machine learning [12, 23, 34, 46], as well as many other applications. Tensor sparsity arises as a result of numerical zeros in the tensors (e.g., due to a negligible interaction as a result of physical distance between particles), or due to not all tensor entries being observed (for example, in tensor completion [40]). Contraction of sparse tensors poses a computational challenge, due to the plethora of possible contractions and decompositions for tensors with 3 dimensions or more.

Acceleration of sparse tensor algebra has been pursued via runtime libraries like Cyclops Tensor Framework (CTF) [57], Tensor Contraction Library (TCL) [58], TiledArray [9, 10], Fastor [49], libtensor [17], ITensor [18], Local Integrated Tensor Framework (LITF) [27]; code generation frameworks like TACO [31], COMET [62], Tensor Contraction Engine (TCE) [6] and also specialized hardware like ExTensor [21], Tensaurus [59] and Hasco [65]. These prior works have focused on enabling generalized contraction of any number of tensors. Additionally, efficient contraction of two sparse or dense tensors has also received attention, SpMM [32], SpTTM [38], SpTV [68], GEMM-like Tensor-Tensor multiplication [58] and contraction of two sparse tensors (SpTC) [41]. However, in the context of tensor decomposition and completion, all of the most important kernels involve contraction of a single sparse tensor (the input dataset) and many smaller dense tensors (representing the decomposition). Such kernels have a single fixed sparsity pattern, unlike contractions such as sparse matrix multiplication, for which the cost and output sparsity is data dependent (dependent on the position of nonzeros). We leverage the data-independent nature of sparse tensor times tensor network (SpTTN) kernels (defined generally in Section 3), to automatically and efficiently find minimum cost implementations.

Prior works with a focus on high-performance tensor decomposition and completion have introduced efficient and parallel implementations for many SpTTN kernels [13, 28, 36, 37, 48, 56]. Most of these works focus specifically on one or two kernels needed for a particular tensor decomposition, e.g., the matricized Khatri-Rao product (MTTKRP) for CP decomposition [8, 14, 26] or the tensor times matrix chain (TTMc) kernel for Tucker [44, 54]). Even for

a single decomposition, different algorithms often rely on different SpTTN kernels [51]. By developing algorithms and libraries for arbitrary SpTTN kernels, we provide functionality for contraction arising (e.g., as a result of a gradient calculation, described in Section 3) from any decomposition/network consisting of dense tensors.

The main challenge in implementation of an SpTTN kernel is finding the most efficient loop nest. In line with prior work [31], we represent such loop nest as a tree, in which each vertex is a loop and its descendants are the loops contained within it. In Section 4, we show how to enumerate all loop nests (assuming fusion is done wherever possible) for a given SpTTN. Since each loop order for any pair of contracted tensors yields a distinct loop nest, the size of this space grows factorially in the loop nest depth m and exponentially in the number of tensors N. We provide a dynamic programming algorithm to find a cost-optimal loop nest with substantially lower cost, namely $O(N^3 2^m m)$ instead of $O((m!)^N)$. We state the algorithm for a general cost function that can be decomposed according to the loop nest tree structure, then provide specific cost functions to minimize buffer size and cache misses.

The new software framework encompassing these SpTTN kernels, SpTTN-Cyclops, is an extension of the CTF [57] library for sparse/dense distributed tensor contractions. CTF provides routines for mapping sparse or dense tensor data to multidimensional processor grids and redistributing data between any pair of grids. Given a mathematical description of a tensor and a sets of contractions, CTF automatically finds a contraction path (sequence of pairs of tensors to contract) and performs each contraction in parallel on a suitable grid. SpTTN-Cyclops instead simultaneously contracts the sparse tensor with all dense tensors in the tensor network, forgoing construction of large (sparse) intermediate tensors required by the CTF method. This all-at-once contraction method has been shown to be efficient in theory and practice for some specific SpTTN kernels such as MTTKRP [3, 4, 53, 56].

The all-at-once contraction approach allows SpTTN-Cyclops to keep the sparse tensor data in place, and rely on existing CTF routines for communication of the other operands. Locally, each processor must then simply execute a loop nest for a smaller SpTTN of the same type. Our framework leverages the new SpTTN loop nest enumeration and search algorithms to select the best choice of loop nest, which is not possible with any previously existing library. To achieve good performance for the innermost loops, we leverage the Basic Linear Algebra Subroutines (BLAS) [7], whenever possible, and incorporate this into our cost function. A similar technique has been used in Mosaic [5], a sparse tensor algebra compiler that demonstrates the benefits of binding tensor sub-expressions to external functions of other tensor algebra libraries and compilers.

We evaluate our framework against the single node performance of TACO and SparseLNR, and the distributed memory implementation of CTF. We also compare SpTTN-Cyclops with the state-of-the-art specialized implementation of one of the SpTTN kernels (SPLATT [56]). Our results demonstrate that we achieve higher performance or close to SPLATT's specifically tuned implementation of one of the kernels. We outperform all three generalized frameworks (TACO, SparseLNR, and CTF) by orders of magnitude. Across some of the kernels, we achieve speedups in the range of 2 to 100x

when compared to these generalized frameworks. We show strong scaling results in the distributed memory setting using tensors of various dimensions and sparsity. We also enable the computation of some of these kernels on larger tensor inputs for which the other frameworks run out of memory.

2 BACKGROUND

2.1 Tensor Notation

We use calligraphic letters to denote tensors, e.g., $\mathcal{T}$. An order N tensor corresponds to an N-dimensional array. We denote elements of tensors in parenthesis, e.g., $\mathcal{T}(i, j, k, l)$ for an order 4 tensor $\mathcal{T}$. The indices that do not appear in the output tensor are considered to be summed (contracted). We use capitalized letters to denote the dimensions of the respective indices. For example, the dimension of index i in $\mathcal{A}(i, j)$ is denoted as I.

2.2 Tensor Sparsity and Sparse Storage

We use one of the most common ways to store sparse tensors, the Compressed Sparse Fiber (CSF) format [53]. We refer to the total number of nonzero elements of a tensor $\mathcal{T}$ as nnz($\mathcal{T}$). For a sparse tensor $\mathcal{T}$ with d dimensions of size $I_1, \ldots, I_d$, we represent the number of non-zeroes in the kth level of the CSF tree for $\mathcal{T}$ (with the first index being at the root) as nnz$^{(I_1 \cdots I_k)}(\mathcal{T})$. Equivalently, this nonzero count may be obtained by considering the number of nonzeros in a reduced tensor obtained by summing away the remaining modes, i.e., nnz$^{(I_1 \cdots I_k)}(\mathcal{T}) = $ nnz($\mathcal{S}$), where $\mathcal{S}(i_1, \ldots, i_k) = \sum_{i_{k+1}, \ldots, i_d} |\mathcal{T}(i_1, \ldots, i_d)|$.

2.3 Tensor Decomposition and Completion Algorithms

Tensor decomposition [33] and completion [55] refer to the problem of decomposing a tensor into a combination of smaller tensors and estimating missing or incomplete values in a tensor, respectively.

The algorithms for both tensor decomposition and completion focus on a single sparse tensor (the input dataset) and require computations that factorize or update the tensor by contracting it with several smaller dense tensors (representing the decomposition). These computations, which we refer to as *kernels*, account for a significant percentage of the overall execution of an algorithm. They have been the focus of high-performance implementations and are typically available as specialized libraries [13, 28, 36, 37, 48, 56]. We list some of the kernels below and describe their existing implementations in the Section 2.4.

1. A standard approach to compute the Canonical Polyadic (CP) decomposition [30] of a tensor is the alternating least squares (ALS) algorithm. Matricized-Tensor times Khatri-Rao Product (MTTKRP) is a key kernel in computing CP-ALS and is the main bottleneck [8, 14, 26],

$$\mathcal{A}(i, a) = \sum_{j,k} \mathcal{T}(i, j, k) \cdot \mathcal{B}(j, a) \cdot C(k, a). \tag{1}$$

2. For Tucker decomposition [63], the analogous to ALS is the higher-order orthogonal iteration (HOOI) algorithm. The primary

(a) Each pairwise contraction has an independent loop nest.

(b) Vertices i and j are fused across the two pairwise contractions.

(c) Vertices i, j and s are fused across the two pairwise contractions.
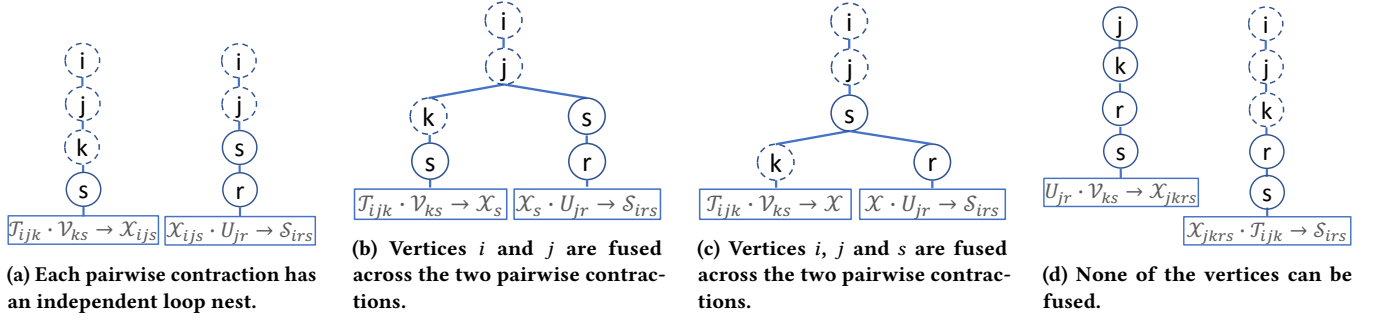
(d) None of the vertices can be fused.

Figure 1: Graphs illustrating loop nests for computing an order 3 TTMc kernel. Sparse loops are shown as dotted vertices.

```
1   T_csf = CSF(T_ijk)
2   X = 0
3   for (i,T_i) in T_csf:
4     for (j,T_ij) in T_i:
5       for (k,t_ijk) in T_ij:
6         for s in range(S):
7           X[i,j,s] += t_ijk * V[k,s]
8   for (i,T_i) in T_csf:
9     for (j,T_ij) in T_i:
10      for s in range(S):
11        for r in range(R):
12          S[i,r,s] += X[i,j,s] * U[j
              ,r]
```

Listing 2: TTMc kernel computed via pairwise contractions.

```
1   T_csf = CSF(T_ijk)
2   for (i,T_i) in T_csf:
3     for (j,T_ij) in T_i:
4       X = 0 // reset intermediate tensor
5       for (k,T_ijk) in T_ij:
6         for s in range(S):
7           X[s] += t_ijk * V[k,s]
8       for s in range(S):
9         for r in range(R):
10          S[i,r,s] += X[s] * U[j,r]
```

Listing 3: TTMc kernel computed using the factorize-and-fuse approach. A single loop nest of *i,j* is used to iterate over both the pairwise contractions.

```
1   T_csf = CSF(T_ijk)
2   for (i,T_i) in T_csf:
3     for (j,T_ij) in T_i:
4       for s in range(S):
5         X = 0 // reset intermediate tensor
6         for (k,T_ijk) in T_ij:
7           X += t_ijk * V[k,s]
8         for r in range(R):
9           S[i,r,s] += X * U[j,r]
```

Listing 4: TTMc kernel computed using the factorize-and-fuse approach. Indices *i,j,s* are fused.

kernel in HOOI is the tensor-times-matrix chain (TTMc) [44, 54],

$$S(i, r, s) = \sum_{j,k} \mathcal{T}(i, j, k) \cdot \mathcal{U}(j, r) \cdot \mathcal{V}(k, s). \quad (2)$$

3. A common generic multi-tensor kernel in tensor completion is the tensor-times-tensor-product (TTTP) [51]. TTTP generalizes the sampled dense-dense matrix multiplication (SDDMM) kernel [11, 43], and is also useful for CP decomposition of sparse tensors,

$$S(i, j, k) = \sum_{r} \mathcal{T}(i, j, k) \cdot \mathcal{U}(i, r) \cdot \mathcal{V}(j, r) \cdot \mathcal{W}(k, r). \quad (3)$$

Note that S has the same sparsity pattern as that of $\mathcal{T}$.

4. Tensor-Times-Tensor-chain (TTTc) kernel used in sparse tensor train decomposition [69] to decompose a higher order sparse tensor using first-order optimization methods,

$$\mathcal{Z}(e, n) = \sum_{i,j,k,l,m,n,a,b,c,d} \mathcal{T}(i, j, k, l, m, n) \cdot \mathcal{A}(i, a) \cdot \mathcal{B}(a, j, b)$$
$$\cdot \mathcal{C}(b, k, c) \cdot \mathcal{D}(c, l, d) \cdot \mathcal{E}(d, m, e). \quad (4)$$

We restrict attention to sparse tensor kernels where the output is dense or has the exact same sparsity as the sparse input tensor. This precludes some common kernels, such as tensor times matrix (TTM) [2] and contraction of two sparse tensors (e.g., SpGEMM [19]), since these generally produce a sparse output.

2.4 Computation of Tensor Kernels in Decomposition and Completion Algorithms

In this section we describe the existing approaches to compute the kernels listed in Section 2.3.

2.4.1 Unfactorized Contraction.

To compute a kernel, we can iterate over all the indices and simultaneously contract all the input tensors in the innermost loop. We refer to this approach as *unfactorized*. This unfactorized loop nest has a depth equal to the number of unique indices. For example, consider an order 3 TTMc kernel in Equation 2. The number of operations is $3 \, \text{nnz}(\mathcal{T}) \cdot R \cdot S$ to leading order. In compiler driven frameworks such as TACO [31] and COMET [62], the schedule generated by default is unfactorized.

The unfactorized approach is optimal in cost for computing certain kernels. For example, the MTTKRP kernel in Equation 1 can be computed using the unfactorized approach with an optimal loop depth of 4. But this approach is asymptotically suboptimal for many other kernels, such as the TTMc.

2.4.2 Pairwise Contraction.

A kernel can be computed with minimal asymptotic complexity (loop depth) by contracting the input tensors pairwise. We refer to this approach as *pairwise contraction*. It is typically used in libraries designed for dense tensor contractions, such as CTF [57], Tensor Computation Engine (TCE) [6], and DEinsum [72]. For example, consider the TTMc kernel in Equation 2. One way in which the tensors can be contracted pairwise is to first contract $\mathcal{T}$ with $\mathcal{V}$, and then its result with $\mathcal{U}$. Each pairwise contraction has an independent loop nest as shown in Listing 2. Both the loop nests have a depth of 4, and the computational cost is $2 \, \text{nnz}(\mathcal{T}) \cdot S + 2 \, \text{nnz}^{(IJ)} \cdot S \cdot R$ to leading order. Even though an unfactorized approach for computing the MTTKRP kernel (Equation 1) has an optimal loop depth, up to a third of the operations can

be eliminated by using pairwise contraction. The unfactorized approach requires $3 \cdot \text{nnz}(\mathcal{T}) \cdot A$ scalar operations, while the pairwise approach requires $2\,\text{nnz}^{(IJK)}(\mathcal{T}) \cdot A + 2\,\text{nnz}^{(IJ)} \cdot A$ operations.

For contractions involving only dense tensors, the pairwise approach can provide an optimal schedule. But for sparse tensors, whose dimensions are often large, this approach can lead to unmanageable memory requirements for storing dense intermediate tensors. In practice, pairwise contraction with sparse storage of such an intermediates has been observed to be much slower than hand-tuned or even unfactorized implementations for SpTTN kernels [51].

2.4.3 Factorize-and-Fuse.
The size of the intermediate tensors can be reduced by loop fusion. Loop nests that share a common index can be nested together with an outer loop that iterates over the shared index. The loop nests that compute the pairwise contractions in Listing 2 can be fused together as shown in Listing 3. We refer to this approach as *factorize-and-fuse*. A single loop nest of *i,j* is used to iterate over both the pairwise contractions and hence the indices are not buffered. The computation cost remains the same as in the pairwise case (in fact, the same set of operations is computed). The size of the intermediate tensor $\mathcal{X}$ is reduced from $I \times J \times S$ to S. Specialized libraries for some of these kernels use a similar approach in their hand-tuned implementations [13, 26, 28, 36, 37, 48, 54, 56].

3 SPTTN KERNELS

In Section 2.3, we listed several kernels for tensor decomposition and completion. We now aim to define these generally. To motivate this definition, consider any tensor decomposition or completion of tensor $\mathcal{T}$ given by a model $\tilde{\mathcal{T}}$ composed of dense tensors $\mathcal{A}_1, \ldots, \mathcal{A}_n$ (factors), the objective function, denoted by f is expressed as,

$$f(\mathcal{A}_1, \ldots, \mathcal{A}_n) = \|\mathcal{T} - \tilde{\mathcal{T}}(\mathcal{A}_1, \ldots, \mathcal{A}_n)\|_F^2.$$

The optimization methods generally leverage all or parts of the gradient of the residual error norm, which yields a contraction of the sparse tensor, with subsets (all but one of the) tensors from the decomposition. The terms involving $\mathcal{T}$ when computing the gradient of f are cost-dominant. Similarly, when computing the residual error (ρ) for tensor completion, which is often employed, e.g., in coordinate descent methods, the terms involving the sparse tensor are cost-dominant. $\rho = \mathcal{T} - \Omega * \tilde{\mathcal{T}}(A_1, \ldots, A_n)$, where $*$ is the Hadamard (pointwise product), the sparse tensor Ω represents the set of observed entries in the input tensor $\mathcal{T}$ and $\tilde{\mathcal{S}}$ is the output tensor obtained by contracting Ω with a network of dense factors.

In general, we define an *SpTTN kernel* as a contraction of a sparse tensor with a set of dense tensors resulting in an output with a dense representation or a sparse tensor with the same sparsity as the sparse input tensor. Hence, in any SpTTN, a subset of the indices in the contraction has a fixed/known sparsity pattern (associated with the input sparse tensor), while the remaining indices iterate only over dense tensors. We generally assume the dense tensors in the SpTTN are fairly small (in comparison to the input sparse tensor).

3.1 Loop Nests and Loop Nest Forests

The computation of a tensor contraction generally involves loop nests of some form. Any loop nest can be represented by an ordered tree or forest, each vertex of which is a loop, and its ordered children are the loop nests contained directly in that loop. Each leaf corresponds to a contraction term (a pair of tensors contracted together). For example, the loop nest in Listing 2 is represented by the tree in Figure 1a. We refer to a tree with a single leaf as a *path graph*. A similar representation is used in TACO [31].

The leaves of the loop nest tree define the order in which the contraction terms are executed. We refer to this order as the *contraction path*. A contraction path for a kernel is valid if we can obtain the output tensor by executing the contraction terms in the order specified by it.

DEFINITION 3.1 (CONTRACTION PATH). *For a contraction of $N + 1$ tensors, a contraction path is given by a depth-first postordering of a $2N + 1$-node binary tree T where the $N + 1$ leaves are the input tensors, and each internal node corresponds to the contraction of a pair of input tensors and/or intermediates, so all non-leaf nodes have exactly two children. We represent a contraction path by the tree T and an ordered collection of index set 3-tuples, $L = (L_1, \ldots, L_N)$, where each L_i contains the indices of the tensor operands and output at each of the N internal tree nodes.*

Note that while a contraction path is defined above based on a tree, this tree is different tree from a loop nest tree. In a loop nest tree, each node corresponds to a loop and each leaf is a term in the contraction path. Hence, a node in the contraction path tree corresponds to a leaf in the loop nest tree. Figure 5(a) shows a contraction path tree for an order 4 TTMc kernel.

(a) (b)
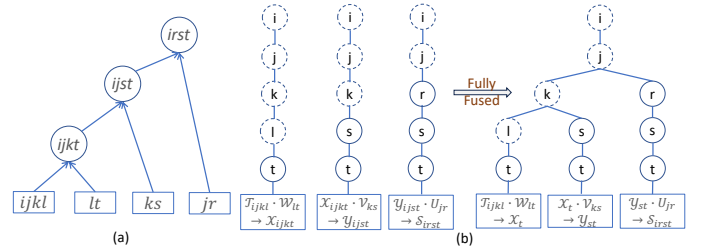
Figure 5: An order 4 TTMc kernel $\mathcal{S}_{irst} = \mathcal{T}_{ijkl} \cdot \mathcal{U}_{jr} \cdot \mathcal{V}_{ks} \cdot \mathcal{W}_{lt}$, where (a) represents the contraction path tree (T) with $L = ((ijkl, lt, ijkt), (ijkt, ks, ijst), (ijst, jr, irst))$, and (b) shows the path graphs corresponding to the contraction path terms, fused to obtain a fully fused loop nest tree.

In a valid loop nest forest, all indices in a contraction term should be loop indices on the path between the corresponding leaf and the tree root, and the path should contain no additional or repeated indices. We refer to this order of loop indices as the *loop order* of the contraction term. For example, in Figure 5(b), the loop order of the first term, $\mathcal{T}_{ijkl} \cdot \mathcal{W}_{lt} \to \mathcal{X}_{ijkt}$, is (i, j, k, l, t). If a vertex has multiple leaves in its subtree, the loop associated with that vertex contains all the contraction terms in that subtree.

DEFINITION 3.2 (LOOP ORDER). *A loop order for a contraction path (T, L), $L = (L_1, \ldots, L_N)$ is defined by an ordered collection*

$A = (A_1, \ldots, A_N)$, where each A_i is an ordered collection of the union of the indices in the 3 index sets contained in L_i.

We say a loop nest tree is fully-fused if no vertex has two consecutive children that correspond to the same index. A fully-fused loop nest and the corresponding tree is obtained by fusion of the path graphs (loop nests) corresponding to each term in A. In Figure 5(b), the path graphs corresponding to the contraction path terms are fused to obtain a fully-fused loop nest tree. A loop nest forest is fully-fused if adding a dummy vertex and connecting it to all roots in the forest yields a fully-fused loop nest tree.

3.2 Intermediate Tensors

Every contraction term except the last, writes its output to an intermediate tensor (buffer). Let the term that generates an intermediate tensor and the term that consumes it be L_x and L_y, respectively. The indices of the intermediate tensor $\mathcal{I}_{L_x L_y}$ are given by

$$\mathrm{inds}(\mathcal{I}_{L_x L_y}) = (\mathrm{inds}(L_x) \cap \mathrm{inds}(L_y)) \setminus S, \tag{5}$$

where S is the set of common ancestors of the two terms in the loop nest graph.

3.3 Contraction Path and Loop Order

The contraction path affects the asymptotic complexity (loop depth) and memory requirements (intermediate tensor sizes) of the computation. For example, consider the various ways to compute the TTMc kernel as shown in Figure 1. In one of the chosen contractions paths, tensors $\mathcal{T}$ and $\mathcal{V}$ are contracted first and the result is then contracted with $\mathcal{U}$. The computation has a maximum loop depth of 4 (Figures 1a, 1b and 1c). A different contraction path of the same kernel, where tensors $\mathcal{U}$ and $\mathcal{V}$ are contracted first and then the result is contracted with $\mathcal{T}$, yields a maximum loop depth of 5 (Figure 1d).

Similarly, for a given contraction path, the ordering of vertices in the path graphs before fusing them, affects the intermediate tensor sizes and other cost metrics of interest. In the previous example of the TTMc kernel, consider the iteration graph in Figure 1a and its fully-fused variant in Figure 1b. Indices i, j and s are common across the two trees in the iteration graph. We are able to fuse vertices i and j but not s (loop order in the first path graph is (i, j, k, s)). This results in an intermediate tensor of size S in Figure 1b (see Listing 3). But if the loop order in the first path graph is (i, j, s, k), we can fuse vertices i, j and s in the iteration graph and obtain a fully-fused loop nest tree with an intermediate tensor of size 1 (scalar) (see Figure 1c and its corresponding loop nest in Listing 4). In the next section, we seek to find cost-optimal loop nests for a given SpTTN kernel, where the cost is defined by a cost model, for example, the intermediate tensor size.

4 FINDING OPTIMAL SPTTN KERNELS

To determine an efficient loop nest for an SpTTN kernel, we first present an approach to enumerate fully-fused trees and later present efficient algorithms to find an optimal tree for simple cost metrics.

4.1 Enumeration of Loop Nests

We seek to find cost-optimal loop nests for a given SpTTN kernel by enumerating only fully-fused loop nest forests and restrict our attention to dense multidimensional buffers (intermediate tensors). We decouple the enumeration into two steps: (1) enumeration of valid contraction paths for a given set of tensors and (2) enumeration of loop orders in the path graphs for a given contraction path.

4.1.1 Enumeration of Contraction Paths.

Let the number of input tensors in the SpTTN be n. To enumerate contraction paths, we employ a function to pick and contract all combinations of two tensors from the list of input tensors. We then recurse over a new list constructed by replacing each pair of contracted tensors with the contraction output. This approach has been studied in the context of finding an optimal contraction path for dense tensor networks [47]. The cost can be analyzed by the recurrence relation, $T(n) = \binom{n}{2} \cdot T(n-1)$ and $T(2) = 1$ (when there are two tensors to contract). The number of valid contraction paths for n tensors is $O\left(\frac{(n!)^2}{n \cdot 2^n}\right)$.

In [24], dynamic programming is used to find the cost-optimal contraction path (tree) given a fixed order of dense tensors to be contracted. This approach is analogous and complementary to our work of finding a cost-optimal loop nest tree for a given contraction path, which we present in Section 4.2.

4.1.2 Enumeration of Loop Orders for a Given Contraction Path.

For a given contraction path, we construct a path graph for each term by picking a loop order for that term. The path graphs are then fused to obtain a fully fused loop nest tree. Each choice of loop order yields a different fused loop nest.

Let the set of indices in the ith term be I_i. The set of indices in the SpTTN is given by $I = \bigcup_{i=1}^{n-1} I_i$. We do an exhaustive search by enumerating all loop orders independently for each path graph and then considering all possible combinations of these orders. Since we do not allow any repeated indices in our path graphs, the loop nests generated in such an enumeration are unique and span all the possible loop nests for a given contraction path. The cardinality of this exhaustive search is given by $\prod_{i=1}^{n-1} |I_i|!$. We later restrict the search to only those loop orders that are consistent with the order of the indices of the sparse tensor, so if a term involves k sparse indices, the number of possible orders for the term is only $|I_i|!/k!$.

4.1.3 Upper Bound on Loop Nests.

For a given SpTTN, the number of loop nests we enumerate has an upper bound given by the product of the number of contraction paths and the number of loop orders for a given contraction path, i.e., $O\left(\frac{(n!)^2 \cdot \prod_{i=1}^{n-1} |I_i|!}{n \cdot 2^n}\right)$. In the following section, we present a dynamic programming algorithm to prune the search space for loop order enumeration.

4.2 Algorithm to Find Cost-Optimal Loop Nests

Enumeration enables autotuning, but for analytic metrics of performance such as buffer size, more efficient search schemes are possible. Different contraction paths yield different fully-fused loop nests, hence we focus our attention to enumeration and search of loop nests for a particular contraction path. In TCE [20, 35], dynamic programming is used to find the cost-optimal loop nest for dense tensor contractions, with one of the cost metrics being the intermediate tensor size. Our efficient search algorithm also

employs dynamic programming, after decoupling order of terms from the tree structure. Given a fixed contraction path order (or a subsequence of the terms, which defines a subproblem), we seek to find a loop nest tree that minimizes a chosen cost metric.

We introduce a peeling primitive for fully fused loop nests to formally define the tree structure. Peeling a fully fused loop nest removes the first outermost loop nest. In a fully-fused loop nest, the outermost loop should iterate over an index that appears in the first contraction, and include within it all subsequent contractions in the contraction path order until one does not include the index.

Definition 4.1 (Peeling of Loop Order). *Given loop order* $A = (A_1, \ldots, A_N)$, *choose* $r \in \{1, \ldots, N\}$ *to be the largest such that* $A_1[1] = A_2[1] = \cdots = A_r[1]$. *Peeling* A *yields two loop orders* $A^{(1)} = (A_1[2:], \ldots, A_r[2:])$ *and* $A^{(2)} = (A_{r+1}, \ldots, A_N)$ *(where* $A_x[2:]$ *denotes the subsequence of all elements in* A_x *except the first element and is omitted if* A_x *has size 1).*

The loop nest tree or forest can then be constructed from the representation $A = (A_1, \ldots, A_N)$ by peeling A iteratively and adding vertices for the two resulting loop orders (if not empty).

Definition 4.2 (Fully-fused Loop Nest Forest). *Given an loop order* $A = (A_1, \ldots, A_n)$, *the corresponding fully-fused loop nest forest* $\mathcal{F}(A) = (V, E)$ *is constructed as follows. Initialize* V *as one vertex corresponding to loop index* $A_1[1]$, *then apply peeling iteratively. At each peeling step, add vertices to* V *for* $A^{(1)}$ *and* $A^{(2)}$ *(unless they are zero-sized) connecting* $A^{(1)}$ *to the vertex representing* A *and* $A^{(2)}$ *to its parent (if any).*

To work with analyzing loop nest forests, it also helps to think about the effect of peeling the loop order on the loop nest tree associated with the loop order.

Definition 4.3 (Peeling of Fully-fused Loop Nest Tree). *Given a loop nest loop order* A *for contraction path* (T, L) *and the corresponding fully-fused loop nest tree* $\mathcal{F}(A) = (V, E)$, *peeling removes the root vertex (index* r*) of the tree. If the root has* k *children, the resulting independent subtrees are associated with loop orders* $B^{(1)}, \ldots, B^{(k)}$, *each of which computes a contraction path for distinct subsets of terms* $L^{(1)}, \ldots, L^{(k)} \subseteq \hat{L}$, *where* $\hat{L}$ *is defined by removing the index* r *from all index sets in* L. *The contraction path tree for the* i*th loop order,* $T^{(i)}$, *is given by removing all vertices from* T *except those corresponding to terms computed in* $L^{(i)}$ *and their children (inputs).*

4.2.1 General Cost Function.

In general, the execution time of a particular fully-fused loop nest tree may depend on architecture or data sparsity in ways that are impractical to fully model and require enumeration and execution. On the other hand, for a simple cost function, e.g., computational cost[1] or intermediate buffer size, the search space can be explored more systematically and efficiently. However, more sophisticated cost functions, which take into account metrics such as cache-efficiency or parallelizability are also of clear interest. We now define a class of functions which we can optimize efficiently, requiring separability of cost according to the structure of the loop nest tree.

---

[1]Since the same contraction path is being considered, all fully-fused loop nest trees have the same asymptotic complexity in tensor size, but order and fusion have an affect on lower-order cost terms.

Definition 4.4 (Tree-separable Cost Function). *Consider a loop nest order* A *for a contraction path* (T, L). *Let* $B^{(1)}, \ldots, B^{(k)}$ *be the loop nest orders for subtrees obtained after peeling root* r *of tree* $\mathcal{F}(A)$ *and* $(T^{(i)}, L^{(i)})$ *be the corresponding contraction path for each* $B^{(i)}$. *A cost function* $f_{\varphi, \oplus}$ *for this loop nest is tree-separable if it satisfies,*

$$f_{\varphi, \oplus}(T, L, A) = \varphi_{T,L,r}\Big(f_{\varphi, \oplus}(T^{(1)}, L^{(1)}, B^{(1)}) \oplus \cdots \oplus$$
$$f_{\varphi, \oplus}(T^{(k)}, L^{(k)}, B^{(k)})\Big),$$

where $\varphi_{T,L,r} : R_+ \to R_+$ *is nondecreasing and* $\oplus$ *is an associative semigroup operator on* R_+ *that is nondecreasing in both variables. If* $\mathcal{F}(A)$ *is a forest,* $f_{\varphi, \oplus}(T, L, A)$ *is given by combining the costs of the independent trees with* $\oplus$.

This definition is quite general as φ is parameterized by the contraction path, and so could be defined at each loop level with full information of the indices/terms involved in the nested loops it contains. At the same time, we observe that f can be evaluated on A recursively, as φ does not depend on all of A, but only the contraction path and the root vertex of $\mathcal{F}(A)$. We could also allow the same parameterization for $\oplus$ without overhead in search complexity, but do not do so for simplicity and due to lack of need.

4.2.2 Maximum Buffer Size.

We now provide a tree-separable cost function to compute the maximum dimension of the intermediate tensors/buffers produced in the execution of a fully fused loop nest. We interchangeably use the terms intermediate tensor and buffer.

Definition 4.5 (Cost Function for Maximum Buffer Dimension). *Consider a fully fused loop nest tree* $\mathcal{F}(A)$ *for loop order* A *with contraction path* (T, L), *where* $T = (V, E)$. *Let* $B^{(1)}, \ldots, B^{(k)}$ *be the loop nest orders for subtrees obtained after peeling* $\mathcal{F}(A)$ *and* $(T^{(i)}, L^{(i)})$ *be the corresponding contraction path for each* $B^{(i)}$. *Let* $Z \subseteq E$ *be the set of edges in the contraction path (oriented towards the root) connecting a node that corresponds to a term* $L_u \in B^{(i)}$ *to another,* $L_v \in B^{(j)}$ *with* $i \neq j$. *The maximum buffer dimension used in the fully fused loop nest is given by* $f_{\varphi, \max}(T, L, A)$ *where* $f_{\varphi, \max}$ *is a tree-separable cost function defined as* $\varphi_{T,L,r}(x) = \max(\rho(T, L, r), x)$, *with* $\rho(T, L, r) = \max_{(L_u, L_v) \in Z, L_u = (K_1, K_2, K_3)} |K_3|$.

The above function is tree-separable since $\varphi_{T,L,r}$ and max satisfy the properties in Definition 4.4 and because Z (and consequently φ) depends only on T, L, r and not on the rest of A. This metric accurately computes the maximum buffer dimension passed through the root loop nest ($\rho(T, L)$), since the size of any buffer used in the fully fused loop nest tree is determined by the indices not yet iterated over (Equation 5), namely those in K_3. Further, since $\oplus$ is a max operator, the maximum buffer dimension needed within any inner loops is also considered by f in a recursive manner. This model can be modified to account for buffer size instead of dimension, by changing $r(A)$ to be the product of the dimensions of the indices in K_3.

4.2.3 Total Number of Cache Misses.

To compute cost as the total number of cache misses for a given contraction path, we consider a simple cache model where the cache can hold N subtensors of size I^D, where I is the tensor dimension

size and $N < I$. For example, if $D = 1$ and if the same column or row of a matrix is accessed consecutively, we assume the column or row is kept in cache. We then model the number of cache misses incurred within each loop, by taking into any misses in contained (inner) loops and counting the number of tensors (inputs and outputs/intermediates computed) that are indexed by the loop index of this loop and still have at least D other indices that need to be iterated over. For each such tensor, at least I^D distinct data from this tensor is loaded in each iteration of the loop, which incurs 1 cache miss. Note that each cache miss in this model is associated with moving I^D tensor data between memory and cache.

DEFINITION 4.6 (COST FUNCTION FOR TOTAL NUMBER OF CACHE MISSES). *Consider a fully fused loop nest tree $\mathcal{F}(A)$ for loop order A with contraction path (T, L). Given a cache of size I^D, the number of cache misses is modeled by $f_{\varphi,+}(T, L, A)$, where $f_{\varphi,+}$ is a tree-separable cost function defined using $\varphi_{T,L,r}(x) = I(r)(\tau(T, L, r) + x)$, where $I(r)$ is the dimension of the root index r and*

$$\tau(T, L, r) = |S|,$$
$$S = \{v : v \in (v_1, v_2, v_3) = L_u, \forall L_u \in L,$$
$$\text{s.t. } r \in v \text{ and } |v| > D\}.$$

Again, it is easy to check that the defined cost function is tree-separable by properties of $\varphi_{T,L,r}$ and $+$. The cost function accurately captures the proposed cache miss model by multiplying the number of cache misses incurred in any loop iteration or its sub-loops by the number of loop iterations. This model can be extended to consider other cache sizes, sparsity, multiple levels of cache, and cache line size.

Algorithm 1 provides a fast search algorithm to find a cost optimal order for tree-separable cost functions. In the pseudocode of the algorithm, for brevity, we use notation such as $T \setminus L_1$ to denote the tree obtained by removing the vertex in the contraction tree T associated with the contraction term L_1. We also use $[x, Y]$ to describe an item or list x being prepended to list Y.

We now provide a proof of correctness and show how the subproblems of Algorithm 1 can be memoized to reduce its complexity. For both, it is helpful to enumerate the subproblems (calls to function ORDER) in terms of

(1) the subsequence of terms included in the subproblem (size of T and L),
(2) the set of indices excluded from the terms (already iterated over), we refer to this set as S.

We use induction on the size of these subproblems to prove correctness.

THEOREM 4.7 (PROOF OF CORRECTNESS OF ALGORITHM 1). *Consider a contraction path (T, L) and a tree-separable cost function f specified by $\varphi_{T,L}$ and $\oplus$. ORDER$(T, L, \varphi_{T,L,r})$ (Algorithm 1) returns two loop orders, A and B, for (T, L), so that A has minimal cost $(f_{\varphi,\oplus}(T, L, A))$ among all loop orders for (T, L) and B has minimal cost among all loop orders for (T, L) that yield a loop nest tree $\mathcal{F}(B)$ with a different root than $\mathcal{F}(A)$.*

PROOF. We prove the theorem statement by induction on the size of L. If there are no indices/terms remaining $(L = \emptyset)$, only the null order is valid. By inductive hypothesis, we assume the

Algorithm 1 Algorithm to find cost-optimal loop order for terms in a given contraction path

Global Input: Loop nest cost function f specified for contraction path (T, L) via parameterized scalar function φ and binary operator $\oplus$.
Input: A contraction path (T, L), with $L = (L_1, \ldots, L_N)$, where each L_i is a 3-tuple of index sets and T is a binary contraction tree.
Output: Two loop orders, A and B, for (T, L), so that A has minimal cost $(f_{\varphi,\oplus}(T, L, A))$ among all loop orders for (T, L) and B has minimal cost among all loop orders for (T, L) that yield a loop nest tree $\mathcal{F}(B)$ with a different root than $\mathcal{F}(A)$.

1: **procedure** ORDER(T, L)
2:     $\delta_A \leftarrow \infty$; $\delta_B \leftarrow \infty$; $A \leftarrow \emptyset$; $B \leftarrow \emptyset$
3:     **if** $L = \emptyset$ **then**
4:         **return** $(\emptyset, \emptyset)$
5:     **if** $L[1] = \emptyset$ **then**
6:         **return** ORDER$(T \setminus L_1, L \setminus L_1, \varphi_{T \setminus L_1, L \setminus L_1})$
7:     $(u, v, w) = L_1$
8:     **for** $q \in u \cup v \cup w$ **do**
9:         $\delta_C \leftarrow \infty$; $C \leftarrow \emptyset$
10:         $k \leftarrow \max\limits_{k \in 1, \ldots, N, \text{ s.t. } q \in L_1, \cdots, q \in L_k} k$
11:         **for** $s \leftarrow 1$ **to** k **do**
12:             Let $(T^{(X)}, L^{(X)})$ be the contraction path restricted to the terms $L_1, \ldots, L_s$ with index q removed.
13:             Let $(T^{(Y)}, L^{(Y)})$ be the contraction path restricted to the terms $L_{s+1}, \ldots, L_N$.
14:             $(A^{(X)}, \star) \leftarrow$ ORDER$(T^{(X)}, L^{(X)})$
15:             $(\bar{A}^{(Y)}, \bar{B}^{(Y)}) \leftarrow$ ORDER$(T^{(Y)}, L^{(Y)})$
16:             ▷ If Y tree has q as root index, the resulting tree would be treated as not fully fused, so take second best tree.
17:             **if** $\bar{A}_1^{(Y)}[1] = q$ **then**
18:                 $A^{(Y)} \leftarrow \bar{B}^{(Y)}$
19:             **else**
20:                 $A^{(Y)} \leftarrow \bar{A}^{(Y)}$
21:             ▷ Compute cost of loop order.
22:             $\delta \leftarrow \varphi_{T,L,q}\left(f_{\varphi,\oplus}(T^{(X)}, L^{(X)}, A^{(X)})\right) \oplus f_{\varphi,\oplus}(T^{(Y)}, L^{(Y)}, A^{(Y)})$
23:             ▷ Update lowest cost loop orders
24:             **if** $\delta < \delta_C$ **then**
25:                 $C \leftarrow [[q, A_1^{(X)}], \ldots [q, A_s^{(X)}], A^{(Y)}]$
26:                 $\delta_C \leftarrow \delta$
27:         **if** $\delta_C < \delta_A$ **then**
28:             $\delta_B \leftarrow \delta_A$; $B \leftarrow A$; $\delta_A \leftarrow \delta_C$; $A \leftarrow C$
29:         **else if** $\delta_C < \delta_B$ **then**
30:             $\delta_B \leftarrow \delta_C$; $B \leftarrow C$
31:     **return** (A, B)

theorem statement holds for any subsequence of terms in L and the associated part of T with any subset of indices removed from all terms in L (the set of indices already iterated over contains S). If the theorem statement does not hold, there must exist some order A' for (T, L) with $f_{\varphi,\oplus}(T, L, A') < f_{\varphi,\oplus}(T, L, A)$. Let r be the root of the first tree in $\mathcal{F}(A')$, $B^{(1)}$ be the first tree in the forest $\mathcal{F}(A')$, and $B^{(2)}$ be the remainder of the forest, with and $(T^{(1)}, L^{(1)})$ and $(T^{(2)}, L^{(2)})$ being the associated contraction paths. Since $f_{\varphi,\oplus}$ is

separable, we have that

$$f_{\varphi,\oplus}(T, L, A') = \varphi_{T,L,r}(f_{\varphi,\oplus}(T^{(1)}, L^{(1)}, B^{(1)}))$$
$$\oplus f_{\varphi,\oplus}(T^{(2)}, L^{(2)}, B^{(2)}).$$

Since $(T^{(1)}, L^{(1)})$ and $(T^{(2)}, L^{(2)})$ are contained and smaller (as defined in our inductive hypothesis) than (T, L), Algorithm 1, when considering root vertex r, would return the minimal cost loop order for both subproblems. Further, the cost of A' would be computed correctly on line 22 of the Algorithm. Since the algorithm instead found A to have a lower cost, we have derived a contradiction. Given optimality of A, its trivial to check that the given optimality condition for B is maintained. □

We now consider the execution cost of Algorithm 1, with the cost of each subproblem memoized. For N ordered terms and m total indices, there are $O((m!)^N)$ loop orders (loop nests). Algorithm 1 needs to consider all subsequences of the N terms and all subsets of the m indices, yielding $O(N^2 2^m)$ subproblems. Each subproblem considers all choices of root index and prefixes of terms that contain that index to iterate over. Thus the cost per subproblem is $O(mN)$ and the overall complexity of the algorithm is $O(N^3 2^m m)$.

5 SPTTN-CYCLOPS FRAMEWORK

We build a runtime framework for SpTTN kernels, which searches for cost-optimal loop nests using the methodology/algorithm introduced in Section 4 and executes the resulting loop nests. Specifically, the framework first considers all contraction paths with optimal asymptotic complexity. For each contraction path, we restrict loop orders to those in which the indices of the sparse tensor are iterated over in the order in which they are stored in the CSF tree. We select the minimum cost loop nest among these using Algorithm 1. If the framework cannot find a loop nest that fits within the constraints set by the cost model, it iterates over the contraction paths with suboptimal asymptotic complexity until it finds a loop nest that adheres to the constraints. While the framework may use different cost functions and employ autotuning, in the experiments, we use a tree-decomposable cost metric that selects the loop nest with the maximum number of independent dense loops with bounded buffer dimension. This choice is made to use BLAS kernels as much as possible while maintaining a bounded amount of storage.

5.1 Algorithm to Generate and Execute Loop Nests

Given a fully fused loop nest tree, in Algorithm 2 we present a runtime algorithm to generate loop nests and execute the contractions. We represent the tree with a sequence of terms (leaves) and a list per term representing the loop order (vertices). This representation is sufficient for the algorithm to infer the structure of a fully fused loop nest tree. We use Algorithm 2 in two stages. In the first stage, we preprocess the fully fused loop nest tree and add hooks to (1) generate nested loops for the dense indices using metaprogramming, (2) identify independent dense loops that can be offloaded to BLAS like kernels. We also allocate memory for the intermediate tensors in this stage. In the second stage, we compute the kernel by executing the preprocessed fully fused loop nest tree. We check for hooks in Line 2 and offload the computation accordingly.

Algorithm 2 Algorithm to generate loop nests

Input: Sequence of terms that represent the contraction path. Each term is a set of three tensors, inp1, inp2 and op.

Input: Depth initially set to 0.

Output: Loop nest to compute the given kernel.

1: **procedure** LOOP_NEST(sequence_of_terms, depth)
2:   **if** depth = |sequence_of_terms[0].idx_order| **then**
3:     $t \leftarrow$ sequence_of_terms[0]
4:     contract(t.inp1, t.inp2, t.op)
5:   idx $\leftarrow$ sequence_of_terms[0].idx_order[depth]
6:   buf_terms $\leftarrow \emptyset$
7:   **for** $c \in$ sequence_of_terms **do**
8:     **if** idx = c.idx_order[depth] **then**
9:       buf_terms $\leftarrow$ buf_terms $\cup$ c
10:     **else**
11:       **if** |buf_terms| ≥ 1 **then**
12:         **for** $i \leftarrow 1, |$buf_terms$|$ **do**
13:           $b \leftarrow$ buf_terms[i]
14:           reset $\leftarrow$ True
15:           **for** $j \leftarrow i+1,$ buf_terms **do**
16:             **if** b.op = buf_terms[j].inp1 or b.op = buf_terms[j].inp2 **then**
17:               reset $\leftarrow$ False
18:           **if** reset = True **then**
19:             b.op $\leftarrow 0$
20:         ▷ generate a loop for idx
21:         LOOP_NEST(buf_terms, depth + 1)
22:       buf_terms $\leftarrow \emptyset$
23:       idx $\leftarrow c$.idx_order[depth]
24:   **if** |buf_terms| ≥ 1 **then**
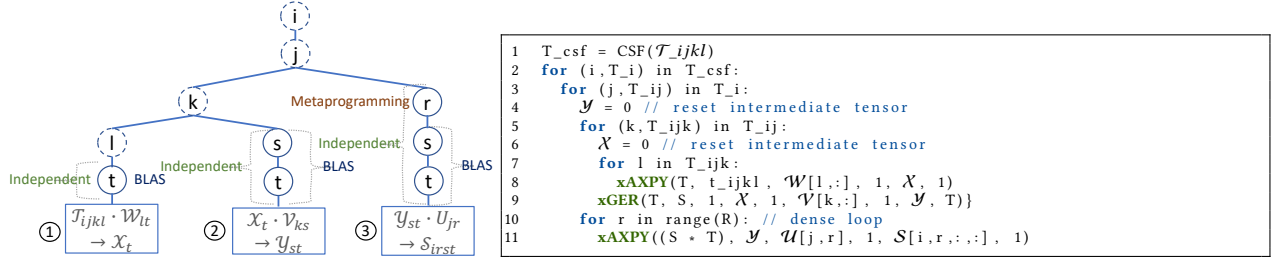25:   ▷ generate a loop for idx
26:     LOOP_NEST(buf_terms, depth + 1)

5.2 Data Distribution

We leverage CTF's [57] data distribution strategy, which uses a cyclic data layout on multidimensional processor grids to achieve load balance and scalability for sparse tensor computations. We continue to hold the main sparse tensor in the same layout for the entire duration of the execution. Each dimension of the tensor is distributed across the processor grid in a cyclic fashion. We redistribute the dense tensors, including the output tensor (if it is dense), along the dimensions it shares with the sparse tensor. Let $\{i_1, \ldots, i_r\}$ be the indices of a dense tensor $\mathcal{D}$ with dimensions $I_1 \times \ldots \times I_r$. Assume a single index of $\mathcal{D}$, i_k, is shared with the sparse tensor. Let the processor grid be $P_1 \times \ldots \times P_n$ and assume i_k is mapped to P_j. Then, D is partially replicated so that all processors $q_1, \ldots, q_j$ with a fixed index q_j own all elements of $\mathcal{D}$, or which $i_k \equiv q_j \mod P_j$. Note that in tensor decomposition and completion algorithms these replicated dimensions are often relatively small. Each processor can now perform local kernel computation without any further data exchange. After the computation we reduce the output tensor and redistribute it to its original mapping on the processor grid.

```
1   T_csf = CSF(𝒯_ijkl)
2   for (i,T_i) in T_csf:
3     for (j,T_ij) in T_i:
4       𝒴 = 0 // reset intermediate tensor
5       for (k,T_ijk) in T_ij:
6         𝒳 = 0 // reset intermediate tensor
7         for l in T_ijk:
8           xAXPY(T, t_ijkl, 𝒲[l,:], 1, 𝒳, 1)
9         xGER(T, S, 1, 𝒳, 1, 𝒱[k,:], 1, 𝒴, T)}
10      for r in range(R): // dense loop
11        xAXPY((S * T), 𝒴, 𝒰[j,r], 1, 𝒮[i,r,:,:], 1)
```

Figure 6: Loop nest for an order 4 TTMc kernel. Loop r of contraction ③ is not via recursion but is generated as a loop by metaprogramming. Contractions ① and ③ are offloaded to BLAS-1, and contraction ② is offloaded to a BLAS-2 kernel.

5.3 Example SpTTN Execution

In Figure 6, we show a fully fused loop nest for the order 4 TTMc kernel, $\mathcal{S}(i,r,s,t) = \sum_{j,k,l} \mathcal{T}(i,j,k,l) \cdot \mathcal{U}(j,r) \cdot \mathcal{V}(k,s) \cdot \mathcal{W}(l,t)$.

6 RELATED WORK

General tensor algebra compilers: TACO [31] and COMET [62] consist of Domain Specific Language (DSL) compilers to generate kernels for both sparse and dense tensors. The default schedules of these frameworks are unfactorized and can be suboptimal for SpTTN kernels.

SparseLNR [16] and ReACT [71] extend TACO and COMET, respectively, with kernel distribution/fusion to support the factorize-and-fuse approach. The contraction path and loop orders for these loop nests are user-specified. Our main contribution is in fully enumerating the space of loop nests and finding a cost-optimal schedule automatically. Furthermore, in our evaluation (in Section 7), we show that SpTTN-Cyclops outperforms SparseLNR by orders of magnitude. For example, across various input tensors considered, SpTTN-Cyclops outperforms SparseLNR by 1.3x to 3.4x and 4x to 110.5x on MTTKRP and TTMc kernels, respectively.

Auto-scheduler: Tensor Contraction Engine (TCE) [6] automatically generates sequence of tensor contractions that minimize intermediate tensor sizes. It primarily focuses on dense tensor operations that are common in quantum chemistry computations. The dynamic programming approach in TCE [20, 35] adopts a bottom-up approach i.e., to find an optimal loop structure, the subtrees of the loop nest tree are evaluated first and memoized. Subsequently, at the root node, various loop structures including the possibility of fusing the subtrees are evaluated to pick the optimal loop structure. Furthermore, in TCE, the tree is partitioned into sub-problems by identifying a set of cut-points. There can be multiple cut-points at a given level. In SpTTN-Cyclops, at any given iteration, we split the problem into two sub-problems, i.e., only the first cut-point is considered, and the cost of the sub-problems is memoized. So a subproblem is a choice for the root index and prefixes of terms that contain that index to iterate over. This approach of SpTTN-Cyclops reduces the cost (for finding an optimal loop nest) when compared to choosing an index for each subtree at a given level and translates into better search complexity.

Protocolized Concrete Index Notation (CIN-P) [1], proposes an automated scheduler that enumerates every schedule of minimum depth and relies on the kernel being small. CIN-P focuses solely on asymptotic costs and CIN-P for TACO discards schedules involving

intermediate tensors of more than one dimension. SpTTN-Cyclops on the other hand tunes over both contraction path and loop orderings. WACO [64] co-optimizes the format and schedule of sparse tensor kernels using a sparse convolutional neural network to model and predict the runtime performance based on the sparsity patterns, formats, and schedules. SparseAuto [15] prunes the search space of schedules for sparse tensor contractions based on both time and intermediate tensor memory requirements. It uses Satisfiability Module Theory (SMT) solvers to pick the smallest number of possible schedules based on user-defined constraints. In CoNST [50], the authors use a constraint-based approach with a Z3 SMT solver to optimize schedules for sparse tensor contractions.

Inspector-executor models incorporated in the compiler transformation frameworks such as Sparse Polyhedral Framework (SPF) [60, 61] enable optimization of sparse computations. In [70], the authors extend SPF to generate optimized sparse tensor codes. They focus on kernels that handle multiple sparse tensors and not SpTTN kernels.

General distributed-memory frameworks: DISTAL [66] extends TACO to target distributed systems. SpDISTAL [67] adopts single-node transformations of TACO and extends DISTAL with new constructs for describing distributions of sparse tensors. SpDISTAL inherits the limitations of TACO in terms of finding an optimal code path for SpTTN kernels. Also, our framework provides automatic distributed memory parallelization without any user intervention. Deinsum [72] provides automatic distributed-memory parallelization of operations on dense tensors. TiledArray [9, 10] is a distributed-memory framework for block-sparse tensors.

Specialized library implementation for SpTTN kernels: SPLATT [56] provides an optimized implementation of MTTKRP on shared and distributed memory systems. GigaTensor [26] implements MTTKRP as a series of Hadamard products and uses the MapReduce paradigm. A parallel algorithm for TTMc which leverages multiple CSF representations is proposed in [54]. Parallel Tensor Infrastructure (ParTI!) [37] is a library for sparse tensor operations (including MTTKRP) and tensor decompositions on multicore CPU and GPU architectures. In [39], as part of ParTI!, the authors propose techniques to reorder the sparse tensor to improve the performance of MTTKRP.

7 EVALUATION

All results are collected on the Stampede2 supercomputer. Each node has an Intel Xeon Phi 7250 CPU ("Knights Landing") with

68 cores, 96GB of DDR4 RAM, and 16GB of high-speed on-chip MCDRAM memory. Additionally, we also run our experiments on a single node equipped with an Intel Xeon Silver 4314 processor ("Ice Lake"), which features a 64KB L1 data cache per core, 1MB L2 cache per core, and a 24MB shared L3 cache. The results of these experiments are reported in Figures 9 and 10. In our distributed memory experiments, we use 64 MPI processes per node. We select a loop nest with the maximum independent dense loops by imposing a bound on the intermediate tensor dimension, maintaining it at two. We compare SpTTN-Cyclops with TACO [31] (commit ID 2b8ece4), general sparse pairwise contraction with CTF [57] (v1.5.5, commit ID 36b1f6d), SPLATT [56] (v1.1.0, commit ID 6cb8628) and SparseLNR [16] (branch dev-fuse, commit ID 8fafdd1). We present results for single thread performance comparing with CTF, TACO, SparseLNR and SPLATT. In SparseLNR, we try to use the optimal schedule derived from SpTTN-Cyclops, using its directives for kernel distribution and loop fusion. In TACO, we use the contraction path picked by SpTTN-Cyclops. For distributed memory performance we compare against CTF and SPLATT.
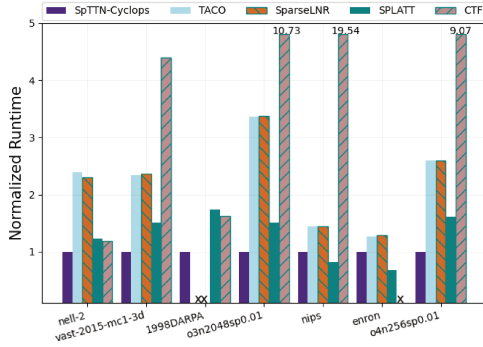

Figure 7: Single thread performance of MTTKRP with $R = 64$.

Datasets: To evaluate the kernels, we use sparse tensors from Formidable Repository of Open Sparse Tensors and Tools (FROSTT) [52] and 1998 DARPA Intrusion Detection Evaluation [25]. For further analysis, we generate random sparse tensors with various dimensions and sparsities. The dense tensors are populated with random data / nonzero positions. If a tensor has identical dimensions, N is used to represent size of the dimensions. A dense tensor shares some of its indices with the sparse tensor. The dimensions of the non-shared indices are denoted using R.

MTTKRP: In Figure 7, we compare the single thread performance of SpTTN-Cyclops to that of the other frameworks. One of the optimal schedules to compute an order 3 MTTKRP in Equation 1 is to have a loop nest that partially contracts $\mathcal{T}$ with $\mathcal{U}$, and then with $\mathcal{V}$. This reduces the number of operations when compared to an unfactorized approach of TACO. SpTTN-Cyclops and SPLATT implement this factorize-and-fuse approach. We observe speedups of 1.3x to 3.4x when compared to TACO. SparseLNR fails to fuse loops for this kernel and has similar performance to TACO. SpTTN-Cyclops achieves speedups of 1.5x to 1.7x, and slowdowns of 0.8x and 0.7x when compared to SPLATT. CTF performs poorly when computing MTTKRP across all tensors. We also conduct strong scaling experiments for MTTKRP on the *nell-2* tensor. Despite being

generic, our approach has performance close to SPLATT, a library optimized for a specific kernel.

TTMc: We observe substantial speedups over TACO. Since we factorize the kernel into pairwise contractions and then fuse loops in SpTTN-Cyclops, there is an asymptotic reduction in computation complexity which translates to these observed speedups. For an order 3 TTMc kernel (Equation 2), SparseLNR generates a schedule that contracts $\mathcal{T}$ with $\mathcal{U}$, and the result with $\mathcal{V}$. Only index i is fused across the two pairwise contractions, and requires an intermediate tensor of $K \times R$ dimension. If the input tensor expression is $\mathcal{S}(i, r, s) = \sum_{j,k} \mathcal{T}(i, j, k) \cdot \mathcal{V}(k, s) \cdot \mathcal{U}(j, r)$, i.e., the position of $\mathcal{U}$ and $\mathcal{V}$ are interchanged in the expression, then SparseLNR defaults to the unfactorized approach of TACO. SpTTN-Cyclops generates a schedule that contracts $\mathcal{T}$ with $\mathcal{V}$, and the result with $\mathcal{U}$. Indices i and j are fused, and the intermediate tensor dimension is S (Listing 3).

For an order 4 TTMc kernel (Section 5.3), SparseLNR generates a schedule that contracts tensors $\mathcal{T}$, $\mathcal{U}$ and $\mathcal{V}$ all-at-once, and the resulting intermediate tensor with $\mathcal{W}$. The intermediate tensor dimension is $L \times R \times S$. Only index i is fused across these two contractions. The maximum loop depth is six. SpTTN-Cyclops generates an asymptotically optimal schedule as shown in Figure 6, which has a maximum loop depth of five.

We are able to run TTMc with TACO and SparseLNR only on two of the considered tensors, *nell-2* and *vast-3d*. On *nell-2*, we observe a speedup of 29.3x and 110.5x over TACO and SparseLNR, respectively. Similarly, in *vast-3d*, we observe a speedup of 125.9x and 4x. We observe speedups over CTF in the range of 0.8x to 12.6x for the tensors considered (we are unable to run TTMc with CTF on *enron* and *nell-2* tensors). On the *nips* tensor where the combination of the imbalanced dimensions of the tensor and the specific value of R does not benefit the fused approach of SpTTN-Cyclops, we see a slowdown of 0.8x. We are unable to execute TTMc on *darpa* using any of the approaches including SpTTN-Cyclops because of the larger memory footprint requirement for the contraction that cannot be accommodated on a single node. In Figures 8(a) and 8(b), we present strong scaling results for MTTKRP and TTMc, respectively. SpTTN-Cyclops outperforms CTF on all node counts, and shows good scaling for both the kernels.

TTTP: We present strong scaling results for TTTP in Figure 8(c). The single node performance of SpTTN-Cyclops over CTF is substantial with over 340x speedup. We observe good scaling for all the considered tensors.

TTTc: In our strong scaling analysis of TTTc, we evaluate two tensors of dimension 80 ($R = 16$) and sparsity at 1% and 0.1%. In both, SpTTN-Cyclops achieves good scaling. SparseLNR generates a default TACO schedule for this kernel. We are unable to run TTTc implementation in TACO and SparseLNR on these kernels with the considered dimensions. However, we generated a smaller tensor with dimensions $N = 40$ and sparsity at 0.1%. SpTTN-Cyclops achieves a speedup of 534x over TACO on it.
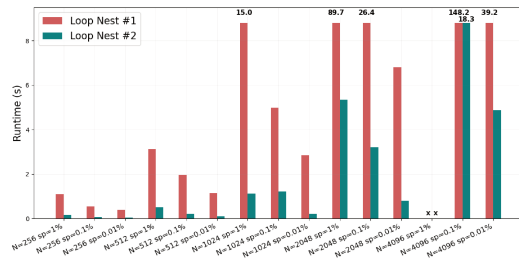
Impact of intermediate tensor dimension: Consider an order 3 all-mode TTMc kernel, $\mathcal{S}(r, s, t) = \sum_{i,j,k} \mathcal{T}(i, j, k) \cdot \mathcal{U}(i, r) \cdot \mathcal{V}(j, s) \cdot \mathcal{W}(k, t)$ (all sparse indices are contracted). The contraction path chosen by SpTTN-Cyclops is $((\mathcal{T}_{ijk} \cdot \mathcal{W}_{kt} \rightarrow \mathcal{X}_{ijt}), (\mathcal{X}_{ijt} \cdot \mathcal{V}_{js} \rightarrow \mathcal{Y}_{ist}), (\mathcal{Y}_{ist} \cdot \mathcal{U}_{ir} \rightarrow \mathcal{S}_{rst}))$. For the chosen contraction path, if we consider a bound of two on the intermediate

(a) TTMc   (b) MTTKRP   (c) TTTP
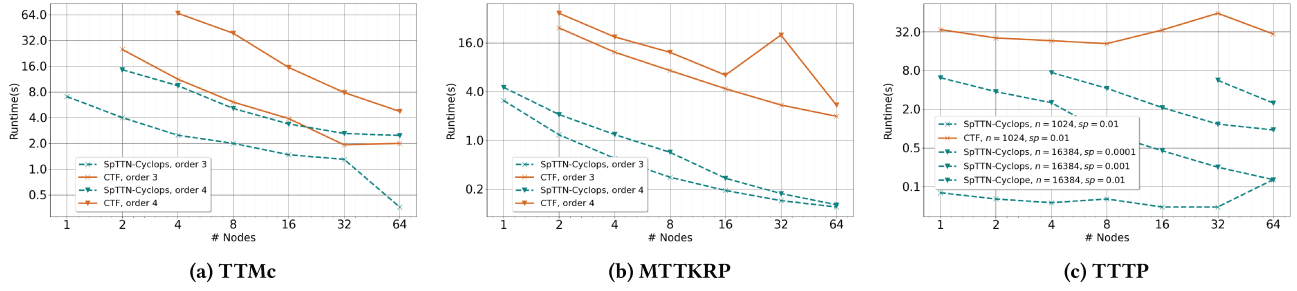
Figure 8: Strong scaling of kernels TTMc, MTTKRP and TTTP. The sparse tensor dimensions are identical across all modes. TTMc and MTTKRP are computed on order 3 and order 4 tensors of 0.1% **sparsity. Their dimensions are set to** 8192 **and** 1024**, respectively. TTTP is computed on order 3 tensors.** $R = 32$**.**

tensor dimension, the loop nest generated by SpTTN-Cyclops, $((i, j, k, t), (i, j, s, t), (i, r, s, t))$, has intermediate tensors $\mathcal{X}$ of size T and $\mathcal{Y}$ of size $S \times T$. For the same contraction path, if we consider a bound of one on the intermediate tensor dimension, the loop nest generated, $((i, t, j, k), (i, t, j, s), (i, t, r, s))$, has intermediate tensors $\mathcal{X}$ of size 1 (scalar) and $\mathcal{Y}$ of size S. In Figure 9, we show the single thread performance of the two loop nests generated by SpTTN-Cyclops for the order 3 all-mode TTMc kernel. We observe that the loop nest with intermediate tensors of size T and $S \times T$ performs better than the loop nest with intermediate tensors of size 1 and S, despite having a larger memory footprint. The contractions in Loop Nest #2 are offloaded to two xAXPY (BLAS-1) (manually-implemented) and one xGER (BLAS-2) kernels. Loop Nest #1, on the other hand, employs an innermost sparse loop to compute the intermediate tensor $\mathcal{X}$. Consequently, only two BLAS kernels are used in this loop nest: one for computing $\mathcal{Y}$ and the other for $\mathcal{S}$.

Impact of loop order: For the order 3 all-mode TTMc kernel and the contraction path chosen by SpTTN-Cyclops, we randomly select 25% of all possible loop orders that have the sparse indices iterated over in the order in which they are stored in the CSF tree. In Figure 10, we show the single thread performance of these randomly picked loop orders. In the loop order picked by SpTTN-Cyclops the intermediate tensors are within a considerable memory bound and also allows for the maximum use of BLAS kernels.

Figure 9: Single thread performance of an order 3 all-mode TTMc contraction. Loop Nest #1 has a bound of 1 and Loop Nest #2 has a bound of 2 on the intermediate tensor dimension. $R = 64$**.**

8 CONCLUSION AND FUTURE WORK

Favorable performance of SpTTN-Cyclops in comparison to other general tensor contraction libraries, as well as comparisons to
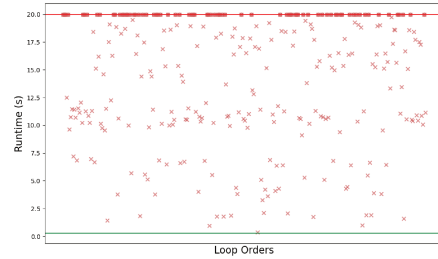
Figure 10: Single thread performance of an order 3 all-mode TTMc contraction with $N = 1024$**,** $R = 32$**, and sparsity at** 0.1% **using randomly picked loop orders. The red line represents the cut-off and the green line represents the runtime of the loop order picked by SpTTN-Cyclops.**

specialized codes, demonstrate that implementation of high-performance SpTTN kernels of interest to tensor decomposition and completion can be effectively automated. As opposed to prior frameworks for sparse tensor contractions, by restricting consideration to a single sparsity pattern and dense buffers, we are able to enumerate and efficiently find the minimum cost SpTTN loop nest. At the same time, the resulting implementations are practical, as they may be accelerated by standard BLAS libraries, and match the structure of existing optimized codes specialized to particular SpTTN contractions. Our framework and evaluation of SpTTN kernels can be extended in several ways. For example, the search space can be extended to include partially-fused loop nests, which may offer additional parallelism.

REFERENCES

[1] Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3519939.3523442

[2] Brett W. Bader and Tamara G. Kolda. 2008. Efficient MATLAB Computations with Sparse and Factored Tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231. https://doi.org/10.1137/060676489

[3] G. Ballard, N. Knight, and K. Rouse. 2018. Communication Lower Bounds for Matricized Tensor Times Khatri-Rao Product. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 557–567. https://doi.org/10.1109/IPDPS.2018.00065

[4] Grey Ballard and Kathryn Rouse. 2020. General Memory-Independent Lower Bound for MTTKRP. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing (PP)*. SIAM, 1–11. https://doi.org/10.1137/1.9781611976137.1

[5] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *Proc. ACM Program. Lang.* 7, PLDI, Article 122 (jun 2023), 26 pages. https://doi.org/10.1145/3591236

[6] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, Xiaoyang Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, Chi chung Lam, Qingda Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. 2005. Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models. *Proc. IEEE* 93, 2 (2005), 276–292. https://doi.org/10.1109/JPROC.2004.840311

[7] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[8] Zachary Blanco, Bangtian Liu, and Maryam Mehri Dehnavi. 2018. CSTF: Large-Scale Sparse Tensor Factorizations on Distributed Platforms. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP 2018)*. Association for Computing Machinery, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/3225058.3225133

[9] Justus A. Calvin, Cannada A. Lewis, and Edward F. Valeev. 2015. Scalable Task-Based Algorithm for Multiplication of Block-Rank-Sparse Matrices. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms* (Austin, Texas) *(IAˆ3 '15)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2833179.2833186

[10] Justus A. Calvin and Edward F. Valeev. 2023. TiledArray: A general-purpose scalable block-sparse tensor framework. https://github.com/valeevgroup/tiledarray

[11] John Canny and Huasha Zhao. 2013. Big Data Analytics with Small Footprint: Squaring the Cloud. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Chicago, Illinois, USA) *(KDD '13)*. Association for Computing Machinery, New York, NY, USA, 95–103. https://doi.org/10.1145/2487575.2487677

[12] Xiaochun Cao, Xingxing Wei, Yahong Han, and Dongdai Lin. 2014. Robust face clustering via tensor decomposition. *IEEE transactions on cybernetics* 45, 11 (2014), 2546–2557.

[13] Jee Choi, Xing Liu, Shaden Smith, and Tyler Simon. 2018. Blocking Optimization Techniques for Sparse Tensor Computation. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 568–577. https://doi.org/10.1109/IPDPS.2018.00066

[14] Joon Hee Choi and S. Vishwanathan. 2014. DFacTo: Distributed Factorization of Tensors. In *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger (Eds.), Vol. 27. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2014/file/d5cfead94f5350c12c322b5b664544c1-Paper.pdf

[15] Adhitha Dias, Logan Anderson, Kirshanthan Sundararajah, Artem Pelenitsyn, and Milind Kulkarni. 2024. SparseAuto: An Auto-Scheduler for Sparse Tensor Computations Using Recursive Loop Nest Restructuring. arXiv:2311.09549 [cs.PL]

[16] 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

[17] Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I. Krylov. 2013. New implementation of high-level correlated methods using a general block-tensor library for high-performance electronic structure calculations. *Journal of Computational Chemistry* (2013).

[18] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. 2022. The ITensor Software Library for Tensor Network Calculations. *SciPost Phys. Codebases* (2022), 4. https://doi.org/10.21468/SciPostPhysCodeb.4

[19] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. 2022. A Systematic Survey of General Sparse Matrix-Matrix Multiplication. *Comput. Surveys* (nov 2022). https://doi.org/10.1145/3571157

[20] Xiaoyang Gao, Sriram Krishnamoorthy, Swarup Kumar Sahoo, Chi-Chung Lam, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan. 2007. Efficient search-space pruning for integrated fusion and tiling transformations: Research Articles. *Concurr. Comput. : Pract. Exper.* 19, 18 (dec 2007), 2425–2443.

[21] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[22] 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.

[23] Edward Hutter and Edgar Solomonik. 2023. Application Performance Modeling via Tensor Completion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (<conf-loc>, <city>Denver</city>, <state>CO</state>, <country>USA</country>, </conf-loc>) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 65, 14 pages. https://doi.org/10.1145/3581784.3607069

[24] Cameron Ibrahim, Danylo Lykov, Zichang He, Yuri Alexeev, and Ilya Safro. 2022. Constructing Optimal Contraction Trees for Tensor Network Quantum Circuit Simulation. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–8. https://doi.org/10.1109/HPEC55821.2022.9926353

[25] Inah Jeon, Evangelos E. Papalexakis, U Kang, and Christos Faloutsos. 2015. HaTen2: Billion-scale tensor decompositions. In *2015 IEEE 31st International Conference on Data Engineering*. 1047–1058. https://doi.org/10.1109/ICDE.2015.7113355

[26] U. Kang, Evangelos Papalexakis, Abhay Harpale, and Christos Faloutsos. 2012. GigaTensor: Scaling Tensor Analysis up by 100 Times - Algorithms and Discoveries. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Beijing, China) *(KDD '12)*. Association for Computing Machinery, New York, NY, USA, 316–324. https://doi.org/10.1145/2339530.2339583

[27] Daniel Kats and Frederick R Manby. 2013. Sparse tensor framework for implementation of general local correlation methods. *The Journal of Chemical Physics* 138, 14 (2013), 144101.

[28] Oguz Kaya and Bora Uçar. 2015. Scalable sparse tensor decompositions in distributed memory systems. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/2807591.2807624

[29] Venera Khoromskaia and Boris N Khoromskij. 2018. Tensor numerical methods in quantum chemistry. In *Tensor Numerical Methods in Quantum Chemistry*. De Gruyter.

[30] Henk A. L. Kiers. 2000. Towards a standardized notation and terminology in multiway analysis. *Journal of Chemometrics* 14, 3 (2000), 105–122. https://doi.org/10.1002/1099-128X(200005/06)14:3<105::AID-CEM582>3.0.CO;2-I

[31] 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

[32] Penporn Koanantakool, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine Yelick. 2016. Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 842–853. https://doi.org/10.1109/IPDPS.2016.117

[33] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500. https://doi.org/10.1137/07070111X

[34] Nadia Kreimer, Aaron Stanton, and Mauricio D Sacchi. 2013. Tensor completion based on nuclear norm minimization for 5D seismic data reconstruction. *Geophysics* 78, 6 (2013), V273–V284.

[35] Chi-Chung Lam, Daniel Cociorva, Gerald Baumgartner, and P. Sadayappan. 2000. Optimization of Memory Usage Requirement for a Class of Loops Implementing Multi-dimensional Integrals. In *Languages and Compilers for Parallel Computing*, Larry Carter and Jeanne Ferrante (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 350–364.

[36] Jiajia Li, Jee Choi, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2017. Model-Driven Sparse CP Decomposition for Higher-Order Tensors. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1048–1057. https://doi.org/10.1109/IPDPS.2017.80

[37] Jiajia Li, Yuchen Ma, and Richard Vuduc. 2018. ParTI! : A Parallel Tensor Infrastructure for multicore CPUs and GPUs. http://parti-project.org Last updated: Jan 2020.

[38] Jiajia Li, Yuchen Ma, Chenggang Yan, and Richard Vuduc. 2016. Optimizing Sparse Tensor Times Matrix on Multi-core and Many-Core Architectures. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IAˆ3)*. 26–33.

https://doi.org/10.1109/IA3.2016.010

[39] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. 2019. Efficient and Effective Sparse Tensor Reordering. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 227–237. https://doi.org/10.1145/3330345.3330366

[40] Ji Liu, Przemyslaw Musialski, Peter Wonka, and Jieping Ye. 2013. Tensor Completion for Estimating Missing Values in Visual Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 1 (2013), 208–220. https://doi.org/10.1109/TPAMI.2012.39

[41] 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

[42] Igor L Markov and Yaoyun Shi. 2008. Simulating quantum computation by contracting tensor networks. *SIAM J. Comput.* 38, 3 (2008), 963–981.

[43] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan. 2018. Sampled Dense Matrix Multiplication for High-Performance Machine Learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 32–41. https://doi.org/10.1109/HiPC.2018.00013

[44] Sejoon Oh, Namyong Park, Sael Lee, and U Kang. 2018. Scalable Tucker Factorization for Sparse Tensors - Algorithms and Discoveries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1120–1131. https://doi.org/10.1109/ICDE.2018.00104

[45] Román Orús. 2014. Advances on tensor network theory: symmetries, fermions, entanglement, and holography. *The European Physical Journal B* 87, 11 (2014), 1–18.

[46] Ioakeim Perros, Robert Chen, Richard Vuduc, and Jimeng Sun. 2015. Sparse hierarchical tucker factorization and its application to healthcare. In *Data Mining (ICDM), 2015 IEEE International conference on*. IEEE, 943–948.

[47] Robert N. C. Pfeifer, Jutho Haegeman, and Frank Verstraete. 2014. Faster identification of optimal contraction sequences for tensor networks. *Phys. Rev. E* 90 (Sep 2014), 033315. Issue 3. https://doi.org/10.1103/PhysRevE.90.033315

[48] Eric T. Phipps and Tamara G. Kolda. 2019. Software for Sparse Tensor Decomposition on Emerging Computing Architectures. *SIAM Journal on Scientific Computing* 41, 3 (2019), C269–C290. https://doi.org/10.1137/18M1210691 arXiv:https://doi.org/10.1137/18M1210691

[49] Roman Poya, Antonio J. Gil, and Rogelio Ortigosa. 2017. A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics. *Computer Physics Communications* (2017). https://doi.org/10.1016/j.cpc.2017.02.016

[50] Saurabh Raje, Yufan Xu, Atanas Rountev, Edward F. Valeev, and Saday Sadayappan. 2024. CoNST: Code Generator for Sparse Tensor Networks. arXiv:2401.04836 [cs.PL]

[51] Navjot Singh, Zecheng Zhang, Xiaoxiao Wu, Naijing Zhang, Siyuan Zhang, and Edgar Solomonik. 2022. Distributed-memory tensor completion for generalized loss functions in python using new sparse tensor kernels. *J. Parallel and Distrib. Comput.* 169 (2022), 269–285. https://doi.org/10.1016/j.jpdc.2022.07.005

[52] 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/

[53] 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* (Austin, Texas) *(IAˆ3 '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2833179.2833183

[54] Shaden Smith and George Karypis. 2017. Accelerating the Tucker Decomposition with Compressed Sparse Tensors. In *Euro-Par 2017: Parallel Processing*, Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro (Eds.). Springer International Publishing, Cham, 653–668.

[55] Shaden Smith, Jongsoo Park, and George Karypis. 2016. An Exploration of Optimization Algorithms for High Performance Tensor Completion. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 359–371. https://doi.org/10.1109/SC.2016.30

[56] Shaden Smith, Niranjan 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. https://doi.org/10.1109/IPDPS.2015.27

[57] 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.

[58] Paul Springer and Paolo Bientinesi. 2018. Design of a High-Performance GEMM-like Tensor–Tensor Multiplication. *ACM Trans. Math. Softw.* 44, 3, Article 28 (Jan 2018), 29 pages. https://doi.org/10.1145/3157733

[59] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[60] 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. https://doi.org/10.1109/JPROC.2018.2857721

[61] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An approach for code generation in the Sparse Polyhedral Framework. *Parallel Comput.* 53 (2016), 32–57. https://doi.org/10.1016/j.parco.2016.02.004

[62] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High Performance Sparse Tensor Algebra Compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 27–38. https://doi.org/10.1109/LLVMHPC54804.2021.00009

[63] L. R. Tucker. 1966c. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31 (1966c), 279–311.

[64] Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman Amarasinghe. 2023. WACO: Learning Workload-Aware Co-optimization of the Format and Schedule of a Sparse Tensor Program. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 920–934. https://doi.org/10.1145/3575693.3575742

[65] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. 2021. Hasco: Towards agile hardware and software co-design for tensor computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1055–1068.

[66] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3519939.3523437

[67] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: Compiling Distributed Sparse Tensor Computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 59, 15 pages.

[68] Wangdong Yang, Kenli Li, and Keqin Li. 2019. A Pipeline Computing Method of SpTV for Three-Order Tensors on CPU and GPU. *ACM Trans. Knowl. Discov. Data* 13, 6, Article 63 (nov 2019), 27 pages. https://doi.org/10.1145/3363575

[69] Longhao Yuan, Qibin Zhao, and Jianting Cao. 2018. High-Order Tensor Completion for Data Recovery via Sparse Tensor-Train Optimization. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 1258–1262. https://doi.org/10.1109/ICASSP.2018.8462592

[70] 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 Trans. Archit. Code Optim.* 20, 1, Article 16 (dec 2022), 26 pages. https://doi.org/10.1145/3566054

[71] 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

[72] Alexandros Nikolaos Ziogas, Grzegorz Kwasniewski, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. 2022. Deinsum: Practically I/O Optimal Multi-Linear Algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 25, 15 pages.