SymProp: Scaling Sparse Symmetric Tucker Decomposition via Symmetry Propagation

Zecheng Li*, Shruti Shivakumar†, Jiajia Li*, Ramakrishnan Kannan‡
*{zli94, jiajia.li}@ncsu.edu, Department of Computer Science, NC State University, Raleigh, USA
†sshivakumar@nvidia.com, NVIDIA Corporation, Santa Clara, USA
‡kannanr@ornl.gov, Oak Ridge National Laboratory, Oak Ridge, TN, USA

***Abstract*—Sparse symmetric tensors are an important class of tensors, and their decompositions serve as powerful tools for revealing low-rank structures. This paper introduces SymProp, a novel approach for scaling sparse symmetric Tucker decomposition by propagating symmetry through intermediate computations. SymProp optimizes two key computational kernels: Sparse Symmetric Tensor Times Same Matrix chain (S^3TTMc) for Higher-Order Orthogonal Iteration (HOOI) and Sparse Symmetric Tensor Times Same Matrix chain Times Core (S^3TTMcTC) for Higher-Order QR Iteration (HOQRI). Our method employs a metaprogramming-based index iteration approach to efficiently handle the upper triangular parts of intermediate dense symmetric tensors. SymProp achieves up to $50.9\times$ speedup over SPLATT and up to $360.8\times$ over Compressed Sparse Symmetric (CSS) format on the S^3TTMc operation. Moreover, our S^3TTMc and S^3TTMcTC implementations support tensor orders four levels higher than state-of-the-art methods. Our HOQRI demonstrates superior scalability and up to a $33.6\times$ speedup over optimized HOOI. By enabling more scalable Tucker decompositions for higher orders, decomposition ranks, and dimension sizes, SymProp opens new possibilities for analyzing complex hypergraph structures in fields such as network science, data mining, and machine learning.**

***Index Terms*—Sparse tensors, symmetric tensors, Tucker decomposition**

I. Introduction

Tensors generalize matrices to higher dimensions, representing multi-dimensional (or high-order) data in numerous applications [1]. Among them, symmetric tensors form an important class of tensors, appearing in diverse fields such as hypergraph analytics [2], [3], deep learning [4], chemistry [5], and data mining [6], [7]. Hypergraphs, for instance, can be represented as sparse symmetric adjacency tensors, with each non-zero entry mapping to a hyperedge [2]. To analyze and compress such tensors, decompositions serve as powerful tools that can reveal low-rank structures for tasks like community detection [3].

Decomposing large-scale tensors presents significant challenges, particularly as tensor order and dimension size increase. Tucker decomposition [8], a fundamental method in tensor analysis, faces a scalability issue as its computational complexity scales exponentially with the tensor order [1]. This phenomenon becomes particularly acute when dealing with sparse symmetric tensors derived from real-world applications, such as hypergraph clustering, which often involve substantially large datasets with high edge degrees, a.k.a., high tensor orders. Tucker decomposition of symmetric adjacency tensors can reveal clustering structures in hypergraphs. However, its lack of scalability for large real-world datasets, especially those with high-order adjacencies, limits its practical application [1], [7]. We observe two inefficiencies in the state-of-the-art sparse symmetric decompositions.

Redundant computation. Existing methods for general sparse tensors [9], [10] often involve redundant computations when applied to symmetric tensors, due to the redundant saving of symmetric entries. Even the newly proposed compact formats specifically for sparse symmetric tensors [11], [12] consider symmetry in the input tensor but fail to exploit symmetry in intermediate data, thus leading to a waste of computational power and memory space. *Our work observes the inherent symmetry property in the intermediate data and leverages it to accelerate sparse symmetric Tucker decompositions.*

Large and high-order data. Current high-performance algorithms for decomposing sparse symmetric tensors are limited in their ability to handle increased input and output data sizes, particularly for high tensor orders and moderate Tucker decomposition ranks [13], which result in large decomposed tensors. By using general sparse tensor formats, like the Compressed Sparse Fibers (CSF) format [9], as the tensor order increases, the memory space quickly runs out, even if they can achieve acceptable runtime performance on small and low-order tensors. While the state-of-the-art Compressed Sparse Symmetric (CSS) format [12] saves computation by using memoization on a tree structure, it still cannot compete with the runtime performance of general sparse formats when the tensor order is low and decomposition ranks become large. *We present performant algorithms to compute symmetric Tucker decomposition of high-order symmetric tensors that scale not only with the tensor size but also with decomposition*

rank.

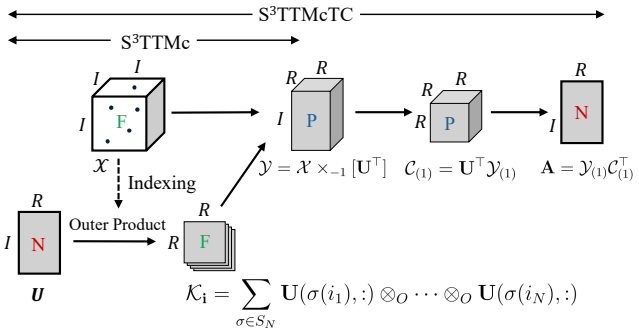


Fig. 1: Overview of symmetry propagation in S^{3}TTMc and S^{3}TTMcTC operations. All tensors, except $\mathcal{X}$, are dense. "N" (No), "P" (Partial), and "F" (Full) denote different symmetric features. $\mathcal{X}$ and $\mathbf{U}$ are the inputs, while $\mathbf{A}$ is the output.

By leveraging and propagating the inherent symmetry in the intermediate data, this paper aims to solve the inefficiencies in computation and memory footprint of existing state-of-the-art approaches which hinder their scaling to large and high-order datasets and decompositions of moderate Tucker rank. We focus on low-rank tensor decomposition with rank below 20, and consider ranks above 4 as *moderate*.

Figure 1 illustrates the symmetry propagation process of two critical computational kernels of Tucker decompositions: Sparse Symmetric Tensor Times Same Matrix chain (S^{3}TTMc) and Sparse Symmetric Tensor Times Same Matrix chain Times Core (S^{3}TTMcTC) from Higher-Order Orthogonal Iteration (HOOI) [13] and Higher-Order QR Iteration (HOQRI) [14] algorithms, respectively. HOOI is widely accepted for Tucker decomposition, while HOQRI employs QR iteration instead of SVD to eliminate large intermediate matrices, potentially accelerating the decomposition process.

In Figure 1, starting from the input sparse symmetric tensor $\mathcal{X}$ and an initialized dense non-symmetric matrix $\mathbf{U}$, the output of S^{3}TTMc is a partially symmetric tensor $\mathcal{Y}$ and a non-symmetric matrix $\mathbf{A}$ for S^{3}TTMcTC. Specifically for S^{3}TTMcTC, it is challenging to utilize the symmetric properties when observing only the input and output (two non-symmetric matrices $\mathbf{U}$ and $\mathbf{A}$). However, our work successfully exploits the symmetric features of intermediate data, such as a series of tensors $\mathcal{K}_{\mathbf{i}}$, $\mathcal{Y}$ and the core tensor $\mathcal{C}$, allowing us to achieve lower computational overhead and reduced memory storage. Based on the three proven properties in Sections III and IV, we develop scalable and efficient symmetry-aware algorithms for both S^{3}TTMc and S^{3}TTMcTC, utilizing the state-of-the-art CSS format for sparse symmetric tensors and a compact format for dense tensors. Our work optimizes these algorithms, ensuring that our improvements are mathematically equivalent and preserve the properties of the original algorithms.

Our contributions are summarized as follows:

- We optimize S^{3}TTMc with compact storage and symmetry-aware computation for intermediate dense symmetric tensors, achieving significant performance improvements for high-order tensors and moderate Tucker ranks. Additionally, we accelerate our algorithm by using efficient index iteration implemented through template metaprogramming. (Section III)
- We optimize S^{3}TTMcTC to fully exploit the intrinsic symmetry in the core tensor. Based on S^{3}TTMc, we enable support for HOQRI with lower complexity. This allows for efficient decomposition of tensors with larger dimension sizes. (Section IV)
- We apply the two optimized kernels to HOOI and HOQRI to significantly enhance their overall performance in tensor decompositions. (Section V)
- We benchmark our implementation to demonstrate the speedups and improved scalability compared to existing methods. Our results show up to a $50.9\times$ speedup and the ability to process tensors with six additional orders compared to SPLATT, as well as up to a $360.8\times$ speedup with tensors of four additional orders compared to CSS. HOQRI outperforms HOOI by up to $33.6\times$ on compatible datasets. Overall, SymProp demonstrates a superior ability to handle tensors with higher order, moderate Tucker rank, and increased dimension sizes compared to previous approaches. (Section VI)

II. Background

TABLE I: Symbol List.

Symbols	Description
$\mathcal{X}, \mathcal{Y}, \mathcal{K}$	Tensors, represented by calligraphic letters.
$\mathcal{X}_{(n)}$	Matricization of $\mathcal{X}$ on n-th mode
$\mathbf{i}, \mathbf{j}$	Index tuple of a non-zero in $\mathcal{X}$. E.g., $\mathbf{i} = (i_1, \ldots, i_N)$
$\mathbf{U}, \mathbf{E}, \mathbf{M}$	Matrices, represented by bold letters.
$\mathcal{X}_f$	IOU representation of dense full symmetric tensor $\mathcal{X}$
$\mathcal{Y}_p$	IOU representation of dense partially symmetric tensor $\mathcal{Y}$
N	Tensor order of $\mathcal{X}$
I	Dimension (mode) size of $\mathcal{X}$
$S_{N,I}$	Size of an order N, dimension size I dense symmetric tensor in compact storage, equals to $\binom{N+I-1}{N}$
$nz(\mathcal{X})$	Non-zero index set of $\mathcal{X}$
$unz(\mathcal{X})$	IOU non-zero index set of $\mathcal{X}$
nnz	Total #Non-zeros of $\mathcal{X}$
$unnz$	Total #IOU non-zeros of $\mathcal{X}$

This section provides an overview of symmetric tensors, their representations, and operations that is related to Tucker decomposition. We introduce the notations used throughout the paper, as summarized in Table I.

A. Symmetric Tensors

Dense Symmetric Tensors. An order-N symmetric tensor $\mathcal{X}$ represents N-dimensional data, and its dimensions are also referred to as modes. Its entries $\mathcal{X}(i_1, i_2, \cdots, i_N)$ (also represented as $\mathcal{X}(\mathbf{i})$ where $\mathbf{i} = (i_1, i_2, \cdots, i_N)$) remain unchanged under any index permutation. For example, consider an order-3, $2 \times 2 \times 2$ symmetric tensor: $\mathcal{T} = \left[\begin{array}{cc|cc} 1 & 2 & 2 & 3 \\ 2 & 3 & 3 & 4 \end{array}\right]$. Entries $(0,0,1)$, $(0,1,0)$, $(1,0,0)$ have the same value 2, and entries

$(0,1,1)$, $(1,0,1)$, $(1,1,0)$ have the same value 3. We refer to an entry $\mathcal{X}_{\mathbf{i}}$ as *index-ordered unique (IOU)* if the index $\mathbf{i}$ is ordered as $i_1 \leq i_2 \leq \cdots \leq i_N$. Only one IOU entry exist per index permutation: $(0,0,1)$ for the value 2 and $(0,1,1)$ for the value 3. Thus, the entry values of the index set $\mathbf{I} = \{(i_1, \ldots, i_N) | i_1 \leq \cdots \leq i_N\}$ represents not only unique but all elements in a symmetric tensor.

We call a tensor *partially symmetric* when its entries remain unchanged under permutations of a subset of its indices. For example, when we state that a tensor has symmetry $\{i_1\}, \{i_2, i_3, i_4\}$ (following the notation from [15]), it means that the tensor is symmetric in the last three dimensions, and its entries remain unchanged when its last three indices are permuted.

Sparse Symmetric Tensors. We only need to store the non-zero values and their corresponding IOU indices for either symmetric or partially symmetric tensors given the data is sparse. The non-zero set of a sparse symmetric or partially symmetric tensor $\mathcal{X}$, denoted as $nz(\mathcal{X})$, is equivalent to having all permutations expanded on the IOU non-zero set $unz(\mathcal{X})$. Our work only involves fully symmetry in sparse tensors.

B. Symmetric Tensor Formats

Dense Symmetric Formats. A simple yet compact storage format [16] for dense (partially) symmetric tensors stores only the IOU entries and arranges them consecutively and linearly in memory, following the lexicographical order of their indices. In the example tensor $\mathcal{T}$, the IOU indices are $(0,0,0)$, $(0,0,1)$, $(0,1,1)$, and $(1,1,1)$ in lexicographical order. The format stores their corresponding entries in a linearized array $[1,2,3,4]$, respectively. This layout stores an order-N symmetric tensor with dimension size I in $\binom{N+I-1}{N} = \frac{(N+I-1)!}{N!(I-1)!}$ entries. Asymptotically, this is $N!$ times fewer than storing all its entries, as demonstrated by the limit: $\lim_{I\to\infty} \frac{I^N}{\binom{N+I-1}{N}} = N!$.

Sparse Symmetric Formats. Various sparse tensor data structures exist for general sparse tensors, such as coordinate-based formats, like COO [1], HiCOO [17], and ALTO [18] and tree-based formats, like CSF [10] and MM-CSF [19]. For sparse symmetric tensors, more compressed formats like UCOO [11] and CSS [11] have been proposed. The UCOO format is the COO format but only storing IOU non-zeros. The Compressed Sparse Symmetric (CSS) format is a tree-based format specifically designed for sparse symmetric tensors, featuring two types of memoization: between IOU non-zeros and within permutations of an IOU non-zero (refer to details in [11]).

We collectively refer to the formats that only save IOU entries as *IOU-based formats*, which include the dense symmetric formats as well as the UCOO and CSS sparse formats.

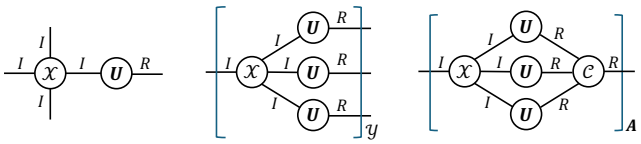
C. Operations

In the following context, we use $\mathcal{X} \in \mathbb{R}^{I \times \cdots \times I}$ to denote a symmetric hypercubical tensor of order N with dimension size I.

1) Tensor Matricization/Unfolding: Tensors can be unfolded into matrices along any mode. The n-th mode unfolding is $\mathcal{X}_{(n)} \in \mathbb{R}^{I_n \times \prod_{i \neq n} I_i}$, defined as: $\mathcal{X}_{(n)}(i_n, j) = \mathcal{X}(i_1, i_2, \ldots, i_N)$, where $j = lin(\mathbf{i} \backslash i_n)$ is the linearized index of $i_1, \ldots, i_N$ excluding i_n.

2) Outer and Kronecker Products: For vectors $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$, their *outer product* $\mathbf{u} \otimes_O \mathbf{v}$ is a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with entries defined as: $A_{ij} = u_i v_j$. When generalizing this operation to tensors, the outer product of multiple vectors results in a tensor of order equal to the number of vectors involved. The *Kronecker product* is closely related to the outer product, but in the *vectorized* form. For vectors $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$, their Kronecker product, denoted as $\mathbf{u} \otimes_K \mathbf{v} \in \mathbb{R}^{mn}$. In other words, the Kronecker product flattens the result of the outer product into a vector, i.e., $\mathbf{u} \otimes_K \mathbf{v} = \text{Vec}\{\mathbf{u} \otimes_O \mathbf{v}\}$.

3) Tensor-Times-Matrix Chain in Tucker Decomposition: We describe operations related to a chain of tensor-times-matrix operations, beginning with an introduction to the tensor-times-matrix operation.



(a) TTM (b) S³TTMc (c) S³TTMcTC

Fig. 2: Tensor diagram notation for tensor-times-matrix and its chains. $\mathcal{X}$ is a symmetric tensor, $\mathbf{U}$ is a matrix.

Tensor-Times-Matrix (TTM). The n-mode tensor-matrix product multiplies a tensor with a matrix along the nth mode. For a symmetric tensor $\mathcal{X}$ and a matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, the product $\mathcal{Y} = \mathcal{X} \times_n \mathbf{U}^\top$ yields a tensor $\mathcal{Y} \in \mathbb{R}^{I \times \cdots \times I \times R \times I \cdots \times I}$. Elementwise: $\mathcal{Y}(i_1, ..., i_{n-1}, r, i_{n+1}, ..., i_N) = \sum_{i_n=1}^{I_n} \mathbf{U}(i_n, r) \mathcal{X}(i_1, ..., i_N)$. Using mode-$n$ unfolding, it can be represented as matrix multiplication:

$$\mathcal{Y}_{(n)} = \mathbf{U}^\top \mathcal{X}_{(n)} \tag{1}$$

Figure 2(a) illustrates the TTM of an order-4 symmetric tensor $\mathcal{X}$ with $\mathbf{U}$ in tensor diagram notation [20]. The nodes represent tensors and tensor indices are notated by edges from each node. The connection implies a multiplication along this mode or index.

Symmetric Tucker Decomposition. For a fixed decomposition rank R, symmetric Tucker decomposition approximates the symmetric tensor $\mathcal{X}$ with a core tensor $\mathcal{C} \in \mathbb{R}^{R \times \cdots \times R}$ and a matrix $\mathbf{U} \in \mathbb{R}^{I \times R}$ with R orthonormal columns. The symmetric Tucker decomposition is formulated as:

$$\min \|\mathcal{X} - \mathcal{C} \times [\mathbf{U}^\top]\|_F^2, \text{ subject to } \mathbf{U}^\top \mathbf{U} = \mathbf{I}.$$

The operation $\mathcal{C} \times [\mathbf{U}^\top]$ is a chain of TTMs that multiplies $\mathbf{U}$ to all modes of $\mathcal{C}$. We will give the definitions of sparse symmetric TTM chain below. The $\|\cdot\|_F$ denotes the Frobenius norm, which is the square root of the sum of the squares of a tensor's elements.

Sparse Symmetric Tensor Times Same Matrix chain (S^{3}TTMc). TTMc applies tensor-matrix products on multiple modes of a tensor. In the popular Tucker decomposition algorithm HOOI [13], TTMc is performed with all modes except one. S^{3}TTMc operation is a special case of the TTMc operation where the tensor $\mathcal{X}$ is symmetric and all matrices are identical. Due to the symmetry of $\mathcal{X}$, the product over all modes except the n-th mode is the same for any n. We conventionally choose the first mode:

$$\mathcal{Y} = \mathcal{X} \times_{-1} [\mathbf{U}^\top] = \mathcal{X} \times_2 \mathbf{U}^\top \times_3 \mathbf{U}^\top \cdots \times_N \mathbf{U}^\top, \quad (2)$$

where $\mathcal{Y} \in \mathbb{R}^{I \times R^{N-1}}$, $\mathbf{U} \in \mathbb{R}^{I \times R}$. The chain of multiplication on an order-4 tensor is visualized in the tensor diagram in Figure 2(b).

For a sparse symmetric tensor $\mathcal{X}$, this matricized $\mathcal{Y}_{(1)}$ could be computed by a sum of Kronecker product of rows in $\mathbf{U}$ indexed by non-zeros.

$$\mathcal{Y}_{(1)}(k,:) = \sum_{\substack{\mathbf{i}=(i_1,\dots i_N)\in nz(\mathcal{X}) \\ i_1=k}} \mathcal{X}(\mathbf{i}) \cdot \left\{ \bigotimes_{j=2}^{N} {}_\kappa \mathbf{U}(i_j,:) \right\} \quad (3)$$

Sparse Symmetric Tensor Times Same Matrix chain Times Core (S^{3}TTMcTC). TTMcTC was first introduced in the HOQRI work for sparse tensors [14], while we consider S^{3}TTMcTC for sparse symmetric case. Mathematically, S^{3}TTMcTC computes the S^{3}TTMc and then multiply it with unfolded core tensor $\mathcal{C}_{(1)}$, resulting in a matrix $\mathbf{A} \in \mathbb{R}^{I \times R}$ for QR decomposition: $\mathbf{A} = \mathcal{Y}_{(1)}\mathcal{C}_{(1)}$. Figure 2(c) illustrates the computation of S^{3}TTMcTC on an order-4 sparse tensor $\mathcal{X}$ using tensor diagram notation.

III. Algorithm Design for S^{3}TTMc

We first discuss the symmtry property that exists in S^{3}TTMc and optimize it. This property will also be used in S^{3}TTMcTC in Section IV.

A. S^{3}TTMc Computation with the IOU-based Format

We represent Equation (3) in an IOU-based format and an outer product pattern in Equation (4) to form a tensor that exhibits a symmetric property, which will be discussed later.

$$\mathcal{Y}_{(1)}(k,:) = \sum_{\substack{\mathbf{i}\in unz(\mathcal{X}) \\ k\in\mathbf{i}}} \mathcal{X}(\mathbf{i}) \ \text{Vec}\left\{\mathcal{K}_{\mathbf{i}\setminus k}\right\}, \quad (4)$$

where

$$\mathcal{K}_{\mathbf{i}} = \sum_{\sigma\in S_N} \mathbf{U}(\sigma(i_1),:) \otimes_O \cdots \otimes_O \mathbf{U}(\sigma(i_N),:), \quad (5)$$

σ is a permutation of $\{1,\dots,N\}$, S_N denotes the symmetric group of all permutations. Then,

$$\text{Vec}\left\{\mathcal{K}_{\mathbf{i}\setminus k}\right\} = \sum_{\sigma\in S_{N-1}} \left\{ \bigotimes_{s=1}^{N-1} {}_\kappa \mathbf{U}(\sigma(j_s),:) \right\}$$

When computing $\mathcal{Y}_{(1)}(i_n,:)$, every permutation $\sigma(i_1),\dots,\sigma(i_{n-1}),\sigma(i_{n+1})\dots\sigma(i_N), \sigma \in S_{N-1}$ of $\mathbf{i}$ excluding i_n contributes to the accumulation.

Take an IOU non-zero $\mathcal{X}_{\mathbf{i}} = 2, \mathbf{i} = (1,3,5)$ in an order-3 tensor $\mathcal{X}$. In the S^{3}TTMc, it adds $2 \times \text{Vec}\{\mathcal{K}_{1,3}\}$ to $\mathcal{Y}_{(1)}(5,:)$ $(k=5)$, $2 \times \text{Vec}\{\mathcal{K}_{1,5}\}$ to $\mathcal{Y}_{(1)}(3,:)$ $(k=3)$, $2 \times \text{Vec}\{\mathcal{K}_{3,5}\}$ to $\mathcal{Y}_{(1)}(1,:)$ $(k=1)$. In general, each IOU non-zero $\mathcal{X}(\mathbf{i})$ contributes $(N-1)!$ terms to the accumulation of rows $\mathcal{Y}_{(1)}(i_1,:),\dots,\mathcal{Y}_{(1)}(i_N,:)$.

B. Symmetry Propagation in the IOU Structure

Intuitively, the $\mathcal{K}$ tensor we construct is fully symmetric due to the formulation with all the index permutations. While each term in the computation may not be symmetric, their sum results in a symmetric tensor $\mathcal{K}$. This property allows us to work with only the unique elements (IOU structure) of each intermediate $\mathcal{K}$ tensor, significantly reducing computational and storage requirements, which we refer to as the **propagation of symmetry**. The following proposition and proof formalizes this intuition, providing the foundation for our SymProp optimization.

Property 1. *The intermediate tensor $\mathcal{K}$ formed by Equation* (5) *is fully symmetric.*

Proof. Given an arbitrary permutation $\tau \in S_N$, we want to show that for an element in $\mathcal{K}$ at index $\mathbf{j} = (j_1,\dots,j_N)$: $\mathcal{K}_{\mathbf{i}}(j_1,\dots,j_N) = \mathcal{K}_{\mathbf{i}}(\tau(j_1),\dots,\tau(j_N))$.

$$\mathcal{K}_{\mathbf{i}}(\tau(j_1),\dots,\tau(j_N)) = \sum_{\sigma\in S_N} \prod_{a=1}^{N} \mathbf{U}(\sigma(i_a),\tau(j_a))$$

Any permutation $\tau \in S_N$ is, by definition, a bijective function from the set $\{1,\dots,N\}$ to itself. Consequently, its inverse τ^{-1} is also a permutation in S_N. Choose a rearrangement of the terms of the product $\mathbf{U}(\sigma(i_1),\tau(j_1))\cdots\mathbf{U}(\sigma(i_N),\tau(j_N))$ using τ^{-1} as the index. Thus, we have

$$\mathcal{K}_{\mathbf{i}}(\tau(j_1),\dots,\tau(j_N)) = \sum_{\sigma\in S_N} \prod_{a=1}^{N} \mathbf{U}(\tau^{-1}(\sigma(i_a)),\tau^{-1}(\tau(j_a)))$$

Let's consider the composition $\pi = \tau^{-1} \circ \sigma$ ($\circ$ represents the function composition). As σ ranges over S_N, π also ranges over all of S_N. Therefore, we can reindex the sum to be:

$$\mathcal{K}_{\mathbf{i}}(\tau(j_1),\dots,\tau(j_N)) = \sum_{\pi\in S_N} \prod_{a=1}^{N} \mathbf{U}(\pi(i_a),j_a) = \mathcal{K}_{\mathbf{i}}(j_1,\dots j_N) \quad (6)$$

The RHS of Equation (6) is identical to the Equation (5), which yields the symmetric property of $\mathcal{K}$ that it is invariant under any permutation of its index. □

SymProp leverages the CSS format, the state-of-the-art format for sparse symmetric tensors, to exploit the symmetry of the input sparse tensor $\mathcal{X}$. We take advantage of its compact format and tree structure for memoization in S^{3}TTMc operations. The symmetry of the intermediate tensor $\mathcal{K}$ is not exclusive to the CSS format. Our approach can be implemented with any IOU-based format, such as UCOO, and is adaptable to future IOU formats.

C. Symmetric Tensor Outer Products

We derive the symmetric tensor outer product formula to compute tensor $\mathcal{Y}$ that takes full advantage of the propagated symmetry, and adopt a metaprogramming technique to address the challenge in index mapping on compact symmetric tensor storage.

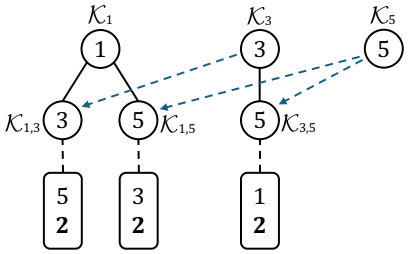


Fig. 3: CSS tree with one IOU non-zero $\mathcal{X}_{\mathbf{i}} = 2, \mathbf{i} = (1, 3, 5)$.

1) Formulation: In CSS format, we utilize tensor outer product to compute the $\mathcal{K}$ tensors in a tree structure to memoize the intermediate results. From CSS tree level $l-1$ to level l, each tensor $\mathcal{K}^{(l)}_{\mathbf{i}}$ at level l is formed by sum of outer products between different $\mathcal{K}^{(l-1)}_{\mathbf{i}\backslash i_n}$s corresponding to the non-zero indices and rows of $\mathbf{U}$, shown in Equation (7). $\mathcal{K} \in \mathbb{R}^{R\times\cdots\times R}$ with the order $|\mathbf{i}| > 1$ equals to the level l.

$$\mathcal{K}^{(l)}_{\mathbf{i}} = \sum_{n=1}^{l} \mathbf{U}(i_n, :) \otimes_O \mathcal{K}^{(l-1)}_{\mathbf{i}\backslash i_n} \tag{7}$$

Figure 3 demonstrates the IOU $\mathbf{i} = (1, 3, 5)$ in the CSS tree structure, decorated by the $\mathcal{K}$ tensors. The tree has two levels, with nodes represented as circles. Level one $\mathcal{K}^{(1)}$s are equal to the corresponding $\mathbf{U}$ rows, while the level two $\mathcal{K}^{(2)}$s are computed using outer products of level one $\mathcal{K}^{(1)}$s with rows of $\mathbf{U}$. The rectangular pairs connected to the second level represent the indices of the $\mathcal{Y}_{(1)}$ rows and their corresponding non-zero values. (For details of the CSS tree structure, refer to [11].)

According to the symmetry propagation in Property 1, we only need to preserve the IOUs in the outer product step when computing each term. Consider a single term in Equation 7, its element-wise representation is

$$\mathcal{K}^{(l)}_{\mathbf{i}}(j_1, \ldots, j_{l-1}, r) \mathrel{+}= \mathbf{U}(i_n, r)\mathcal{K}^{(l-1)}_{\mathbf{i}\backslash i_n}(j_1, \ldots, j_{l-1}). \tag{8}$$

Thus, we can compute $\mathcal{Y}_{(1)}(k, :)$ on only IOU indices via Equation (4).

2) Efficient Index Iteration: Extracting the IOU indices of $\mathcal{K}^{(l)}$ and $\mathcal{K}^{(l-1)}$ during the iteration over their index space is required in Equation (8) to compute the outer product. Since $\mathcal{K}^{(l)}$ at each level is symmetric, we use compact storage format by linearizing IOUs in a lexicographical order mentioned in Section II-B [15] to store it. Thus, a careful algorithm design is needed to obtain good memory locality and avoid the overhead of computing the mapping of multi-dimensional indices to a particular memory location for each IOU entry.

Our approach addresses this challenge by using nested for loops to iterate the index space of $\mathcal{K}^{(l)}$, shown in Algorithm 1. This index iteration matches with the memory layouts of both $\mathcal{K}^{(l)}$ and $\mathcal{K}^{(l-1)}$, leads to continuous memory access. This nested for-loop implementation has two benefits: 1) no computational overhead for the index mapping, which could be $O(N+R)$ in existing work [21], [22]; 2) compiler-friendly, allowing the compiler to automatically optimize the code due to its predictable behavior, fewer branches, and continuous memory access, compared to the existing coupled for- and while-loop approach that iterates indices by tracing back index boundaries for each mode [16]. In Algorithm 1, the loc_l iterates on the $\mathcal{K}^{(l)}$ and increments at the level-l loop while loc_{l-1} iterates on $\mathcal{K}^{(l-1)}$ and shares the same $l-1$ for-loops.

Algorithm 1 Symmetric outer product algorithm for Equation (8).

```
   // Base case: K^(1) = U(i_1, :).
   // Index memory locations of K^(l) and K^(l-1).
1: Initialize loc_l, loc_{l-1} ← 0
2: for i_1 ← 0 to R − 1 do                          ▷ Index Iteration
3:     for i_2 ← i_1 to R − 1 do
4:         ...
5:         for i_{l-1} ← i_{l-2} to R − 1 do
6:             for i_l ← i_{l-1} to R − 1 do
7:                 K^(l)[loc_l] ← U[i_n, i_l] × K^(l-1)[loc_{l-1}]
8:                 loc_l ← loc_l + 1
9:             end for
10:            loc_{l-1} ← loc_{l-1} + 1
11:        end for
12:        ...
13:    end for
14: end for
```

3) Template Metaprogramming: However, the number of nested for-loops in Algorithm 1 is unknown before execution and varies with different level l values. This brings difficulties in implementing this algorithm. We leverage template metaprogramming to generate the nested for-loops for all possible orders at compile time and dynamically dispatched during execution time, avoiding the need to implement them separately.

Our approach utilizes a recursive template function to generate N nested loops, one for each level. It maintains an *indices* array to track the N indices for the right-hand side tensor $\mathcal{K}^{(l-1)}$. It recursively calls the *iterate_* function to allow each nested loop starts from the index of its outer loop. The compiler unrolls and inlines these recursive calls, effectively transforming them into the equivalent for-loop implementation of Line 2-5 in Algorithm 1 at compile time.

```
1  template <typename Func, size_t R>
2  static void iterate_(dim_t dim, Func &&f, array<dim_t, N>
     &indices, index_t &idx, dim_t start = 0) {
3    if constexpr (R == 0) {
4      f(indices, idx);
5      idx++;
6    } else {
7      for (dim_t i = start; i < dim; ++i) {
8        indices[N-R] = i;
9        iterate<Func, R-1>(dim, f, indices, idx, i);
10     }
11   }
12 }
13
14 template <typename Func>
15 static void iterate(dim_t dim, Func &&f) {
```

```
16   std::array<dim_t, N> indices{};
17   index_t oned_idx = 0;
18   iterate_<Func, N>(dim, std::forward<Func>(f), indices,
   oned_idx);
19 }
```

This method allows for flexible computations on iterated elements through a template function parameter. The Line 6-9 in Algorithm 1 is passed as a function parameter in Line 15 of the C++ code and is being called with the IOU and memory indices of $\mathcal{K}^{(l-1)}$. It iterates the symmetric tensor consecutively in memory, keeping minimal state (N indices) and avoiding the runtime cost of index computation. It is highly efficient for operations like outer products that linearly traverse the tensor.

D. Complexity Analysis

Since we store fully symmetric tensors in a compact format, we compute for IOU indices where $i_1 \leq i_2 \leq \cdots \leq i_{l-1} \leq j < R$. This leads to a reduction of computation from $2R^l$ to $2S_{l,R}$ floating-point operations for a single outer product at level-l. Therefore, we reduce the level-l complexity in S^{3}TTMc from $c_{css}(l;N,R) = (2l-1)\binom{N}{l}R^l unnz$ [12] to

$$c_{sp}(l;N,R) = (2l-1)\binom{N}{l}S_{l,R}unnz \tag{9}$$

where *sp* is the shorthand for SymProp. Note that $S_{l,R} = \binom{l+R-1}{l}$, the reduction in level-l computation is $\frac{R^l}{S_{l,R}} = \begin{cases} l! & \text{as } R \to \infty \\ \frac{2^l}{l+1} & \text{when } R = 2 \end{cases}$.

IV. Algorithm Design for S^{3}TTMcTC

Based on the partially symmetric property of the $\mathcal{Y}$ tensor, we propagate the symmetry to derive a computationally efficient S^{3}TTMcTC.

A. Symmetric Propagated S^{3}TTMcTC

Algorithm 2 shows our optimized S^{3}TTMcTC on the CSS format.

Algorithm 2 Optimized CSS-based S^{3}TTMcTC.

Input: Sparse symmetric tensor $\mathcal{X}$, orthonormal factor $\mathbf{U}$
Output: Intermediate matrix $\mathbf{A}$

1: $\mathcal{Y}_p = \mathcal{X} \times_{-1} [\mathbf{U}^\top]$ ▷ *Optimized S^{3}TTMc*
2: $\mathcal{C}_{p(1)} = \mathbf{U}^\top \mathcal{Y}_{p(1)}$ ▷ *Refer to Property 2*
3: $\mathbf{A} = \mathcal{Y}_{(1)}\mathcal{C}_{(1)}^\top = \mathcal{Y}_{p(1)}\mathbf{M}\mathcal{C}_{p(1)}^\top$ ▷ *Refer to Property 3*

Based on Property 1, the $\mathcal{K}$ tensors in Equation (5) are fully symmetric. Since the sum of symmetric tensors is still symmetric, we identify that tensor $\mathcal{Y}$ is symmetric on all dimensions but the first, which we state as partial symmetry $\{i_1\},\{i_2,\ldots,i_N\}$. Recall that a dense symmetric tensor with order N and dimension size R has $S_{N,R} = \binom{N+R-1}{N}$ unique non-zero entries. We store $\mathcal{Y}$ in the compact form $\mathcal{Y}_p$ with $I \cdot S_{N-1,R}$ space. $\mathcal{Y}_{p(1)} \in \mathbb{R}^{I \times S_{N-1,R}}$ unfolds the symmetric modes $\{i_2,\ldots,i_N\}$ to matrix rows. Unlike the HOQRI paper [14], which completely avoids explicitly storing the $\mathcal{Y}$ tensor, we still save it in the compact format $\mathcal{Y}_p$ since storing it is not a bottleneck compared to the space required for the tree format for input and intermediate tensors. Moreover, we avoid redundant computation of CSS format by keeping the $\mathcal{Y}_p$ tensor in memory.

The core tensor $\mathcal{C}$ in symmetric Tucker decomposition is fully symmetric [23]. However, we choose to represent it in partially symmetric form with symmetry $\{i_1\},\{i_2,\ldots,i_N\}$ that matches the memory layout of rows of $\mathcal{Y}_{p(1)}$ and columns of $\mathcal{C}_{p(1)}^\top$. Thus, we can compute a regular matrix multiplication $\mathcal{C}_{p(1)} = \mathbf{U}^\top \mathcal{Y}_{p(1)}$ in Line 2, where $\mathcal{C}_{p(1)} \in \mathbb{R}^{R \times S_{N-1,R}}$, allowing simpler multiplication on the symmetric modes. The correctness of this multiplication is proven in Section IV-B. If $\mathcal{C}$ is stored in the fully symmetric form $\mathcal{C}_f$, multiplying $\mathcal{Y}_{p(1)}\mathcal{C}_{f(1)}^\top$ becomes non-trivial because index mapping is needed, leading to not only implementation complexity but also computational overheads described in Section III-C2. The partially symmetric $\mathcal{C}_p$ allows us to apply the aforementioned index iteration to efficiently iterate over the unique non-zeros, and the memory overhead is negligible since $\mathcal{C}_{p(1)} \in \mathbb{R}^{R \times S_{N-1,R}}$ is much smaller than $\mathcal{Y}_{p(1)} \in \mathbb{R}^{I \times S_{N-1,R}}$. The multiplication on the symmetric modes of $\mathcal{C}_p$ in Line 3 is detailed in Section IV-C.

Algorithm 2 allows us to compute $\mathbf{A}$ in $O(IRS_{N-1,R})$ complexity in addition to S^{3}TTMc.

B. Multiplication on the Non-symmetric Mode

Line 2 in Algorithm 2 forms the unfolded core tensor $\mathcal{C}$ in partially symmetric compact form $\mathcal{C}_p$. By the Tucker decomposition, $\mathcal{C} = \mathcal{X} \times [\mathbf{U}^\top]$ on all modes, here we compute $\mathcal{Y} = \mathcal{X} \times_{-1} [\mathbf{U}^\top]$ with S^{3}TTMc and subsequent $\mathcal{C} = \mathcal{Y} \times_1 \mathbf{U}^\top$.

Property 2. *Let $\mathcal{Y}_p$ be a partially symmetric tensor of order N, with symmetric modes $\{i_2,\ldots,i_N\}$. The tensor-matrix multiplication $\mathcal{Y}_p \times_1 \mathbf{U}^\top$ along the non-symmetric mode i_1 produces a partially symmetric tensor $\mathcal{C}_p$, which retains the same layout as $\mathcal{Y}_p$ for the symmetric modes $\{i_2,\ldots,i_N\}$.*

Proof. Let the size of a dense order-N tensor with dimension size R be R^N and the size of a symmetric tensor with the same shape, stored in compact form, be $S_{N,R}$. The tensor-matrix multiplication $\mathcal{C}_p = \mathcal{Y}_p \times_1 \mathbf{U}^\top$ is equivalent to $\mathcal{C}_{p(1)} = \mathbf{U}^\top \mathcal{Y}_{p(1)}$ based on Equation (1).

We define an expansion matrix $\mathbf{E} \in \mathbb{R}^{R^{N-1} \times S_{N-1,R}}$ that maps from the unfolded compact symmetric rows of $\mathcal{Y}_p$ to their full representation. $\mathbf{E}(j,i) = 1$ if the i-th IOU in the compact form expands to include the j-th linearized full representation index, and 0 otherwise. This map gives $\mathcal{Y}_{(1)} = \mathcal{Y}_{p(1)}\mathbf{E}^\top$. Let's consider the full multiplication:

$$\mathcal{C}_{(1)} = \mathbf{U}^\top \mathcal{Y}_{(1)} = \mathbf{U}^\top(\mathcal{Y}_{p(1)}\mathbf{E}^\top) = (\mathbf{U}^\top \mathcal{Y}_{p(1)})\mathbf{E}^\top = \mathcal{C}_{p(1)}\mathbf{E}^\top$$

Since $\mathcal{C}_{(1)}$ and $\mathcal{Y}_{(1)}$ are constructed from their respective compact forms using the same indicator matrix $\mathbf{E}$, $\mathcal{C}_{p(1)}$ and $\mathcal{Y}_{p(1)}$ have the same symmetric layout. □

Property 2 proves the correctness of Line 2 in Algorithm 2.

C. Multiplication on the Unfolded Symmetric Modes

We now present our approach to the $\mathbf{A} = \mathcal{Y}_{(1)}\mathcal{C}_{(1)}^{\top}$ at Line 3. We perform the matrix multiplication on the partially symmetric representation $\mathcal{Y}_p$ and $\mathcal{C}_p$.

Property 3. *$\mathcal{Y}_{(1)}\mathcal{C}_{(1)}^{\top} = \mathcal{Y}_{p(1)}\mathbf{M}\mathcal{C}_{p(1)}^{\top}$, where $\mathbf{M}$ is the diagonal matrix such that $\mathbf{M}_{ii}$ is the number of permutations of the i-th IOU non-zero obtained from the compact ordering of the dense symmetric tensor $\mathcal{Y}_{p(1)}(q,:)$ for any row q.*

Proof. Since $\mathcal{Y}_{(1)}\mathcal{C}_{(1)}^{\top} = \mathcal{Y}_{p(1)}\mathbf{E}^{\top}\mathbf{E}\mathcal{C}_{p(1)}^{\top}$, $\mathbf{M} = \mathbf{E}^{\top}\mathbf{E}$.

$$\mathbf{M}(i,k) = (\mathbf{E}^{\top}\mathbf{E})(i,k) = \sum_{j=1}^{R^{N-1}} \mathbf{E}(j,i)\mathbf{E}(j,k)$$

Following the definition of $\mathbf{E}$, $\mathbf{E}(j,i) = 1$ if the i-th IOU in the compact form expands to include the j-th linearized full representation index, and 0 otherwise.

When $i \neq k$, the i-th and k-th IOU are different. There does not exist a full tensor index that is a permutation of both IOU at i and k at the same time. Therefore, $\mathbf{M}(i,k) = 0$ for all $i \neq k$.

When $i = k$, $\mathbf{M}(i,i) = \sum_{j=1}^{R^{N-1}} \mathbf{E}(j,i)\mathbf{E}(j,i)$. Since $\mathbf{E}$ is a binary matrix, it is equivalent to $\sum_{j=1}^{R^{N-1}} \mathbf{E}(j,i)$ which counts the number of 1s in the i-th column of $\mathbf{E}$. This sums up to the number of permutations of the i-th IOU of any row in $\mathcal{Y}_{p(1)}$. □

In our implementation, instead of storing $\mathbf{M}$ as a matrix, we represent it as a vector $\mathbf{p} \in \mathbb{R}^{S_{N-1,R}}$, where each p_i corresponds to the i-th diagonal element of $\mathbf{M}$. Therefore, we scale each row of $\mathcal{C}_{p(1)}^{\top} \in \mathbb{R}^{S_{N-1,R},R}$ by the corresponding element of $\mathbf{p}$ (We choose $\mathcal{C}_p$ instead of $\mathcal{Y}_p$ since it is smaller).

$$\mathcal{C}'^{\top}_{p(1)}(i,j) = (\mathbf{M}\mathcal{C}_{p(1)}^{\top})(i,j) = \mathbf{p}(i) \cdot \mathcal{C}_{p(1)}^{\top}(i,j)$$

The subsequent dense matrix multiplication of $\mathcal{Y}_{(1)}$ and the scaled $\mathcal{C}'^{\top}_{p(1)}$ is accelerated by calling the BLAS library.

For an IOU $i_1, \ldots, i_N$ with index value frequencies $k_1, \ldots, k_m$, the number of permutations of the IOU is given by the multinomial coefficient $\binom{N}{k_1,k_2,\ldots,k_m}$ [16]. Before the iterations in HOQRI, we enumerate all the IOU indices of one row in $\mathcal{Y}_{(1)}$ through the index iteration method in Section III-C2, compute frequency for each index, and apply the multinomial coefficient to compute each element in vector $\mathbf{p}$. Since same $\mathbf{p}$ is used in different iterations of Tucker decomposition, we compute it once and memoize it.

V. Sparse Symmetric Tucker decomposition

To improve the scalability of symmetric Tucker decomposition, we focus on two key algorithms: Higher-Order Orthogonal Iteration (HOOI) [13] and Higher-Order QR Iteration (HOQRI) [14]. These algorithms aim to minimize the Tucker objective, which is the least-squares cost function, according to [13]: $f(\hat{\mathcal{X}}) = \|\mathcal{X} - \hat{\mathcal{X}}\|_F^2 = \|\mathcal{X}\|_F^2 - \|\mathcal{C}\|_F^2$ where $\mathcal{C}$ is the core tensor, $\hat{\mathcal{X}} = \mathcal{C} \times_{-1} [\mathbf{U}^{\top}]$ is the Tucker approximation of the original tensor $\mathcal{X}$. Both algorithms initialize the factor matrix $\mathbf{U}$ using Higher-Order Singular Value Decomposition (HOSVD) [24] or randomly. Symmetric HOSVD initialization computes the R leading left singular vectors of the mode-1 unfolding of the tensor to obtain a $\mathbf{U}$ matrix as a starting point.

A. HOOI

The Higher-Order Orthogonal Iteration algorithm [13] is based on the Alternating Least Square technique. We focus on the sparse symmetric HOOI where the same matrix $\mathbf{U}$ is used for each Tucker factor. As shown in Algorithm 3, each iteration performs the S^{3}TTMc operation, followed by updating $\mathbf{U}$ with the left singular vectors via SVD on the unfolded tensor. These steps are repeated until the stopping criterion is met, either when the objective converges or the maximum number of iterations is reached.

Algorithm 3 Order-N sparse symmetric HOOI

Input: Sparse symmetric tensor $\mathcal{X}$
Output: Core tensor $\mathcal{C}$, and orthonormal matrix $\mathbf{U}$
1: Initialize $\mathbf{U} \in \mathbb{R}^{I \times R}$ randomly or using HOSVD
2: **while** $f(\hat{\mathcal{X}})$ not improving or max iteration reached **do**
3: $\quad\mathcal{Y} = \mathcal{X} \times_{-1} [\mathbf{U}^{\top}]$ ▷ *S^{3}TTMc*
4: $\quad\mathbf{U} \leftarrow R$ left singular vectors of $\mathcal{Y}_{(1)}$ via SVD
5: $\quad\mathcal{C} = \mathcal{Y} \times_1 \mathbf{U}^{\top}$
6: $\quad f(\hat{\mathcal{X}}) = \|\mathcal{X}\|^2 - \|\mathcal{C}\|^2$
7: **end while**

B. HOQRI

Higher-Order QR Iteration [14] aims to improve the scalability of Tucker decomposition by using QR decomposition for orthogonalization in each iteration instead of the SVD in HOOI. It also eliminates the large intermediate matrix $\mathcal{Y}$ by introducing the novel TTMcTC kernel. The convergence of HOQRI has been theoretically proven in previous work [14], [25]. Our implementation of S^{3}TTMcTC differs from the original HOQRI's n-ary contraction approach that computes the $\mathbf{A}$ matrix element-by-element without intermediate memory. We leverage S^{3}TTMc and store the $\mathcal{Y}$ tensor in partially symmetric form to achieve lower computational complexity as shown in Section IV.

Algorithm 4 Order-N sparse symmetric HOQRI

Input: Sparse symmetric tensor $\mathcal{X}$
Output: Core tensor $\mathcal{C}$, and orthonormal matrix $\mathbf{U}$
1: Initialize $\mathbf{U} \in \mathbb{R}^{I \times R}$ randomly or using HOSVD
2: **while** $f(\hat{\mathcal{X}})$ not improving or max iteration reached **do**
3: $\quad\mathcal{C} = \mathcal{X} \times [\mathbf{U}^{\top}]$
4: $\quad\mathbf{A} = \text{S}^3\text{TTMcTC}(\mathcal{X}, \mathbf{U})$
5: $\quad\mathbf{U} \leftarrow$ an orthonormal basis of $\mathbf{A}$ via QR
6: $\quad f(\hat{\mathcal{X}}) = \|\mathcal{X}\|^2 - \|\mathcal{C}\|^2$
7: **end while**

Compared to HOOI, HOQRI typically demonstrates better scaling in terms of memory and computation, especially for tensors with higher tensor orders and larger dimension sizes. While HOQRI may require more iterations to converge, each iteration is computationally faster, potentially leading to faster overall solutions. This makes HOQRI particularly suitable

for high-order, high-rank tensor decompositions where HOOI might struggle due to memory constraints or high computational costs.

TABLE II: Tucker decomposition algorithm complexities

Algorithm	Complexity
HOOI-CSS [11]	$C^{CSS} + O(IR^{N-1}\min(I, R^{N-1}))$
HOOI-SymProp	$C^{SP} + O(IR^{N-1}\min(I, R^{N-1}))$
HOQRI [14]	$O(R^N N! unnz)$
HOQRI-SymProp	$C^{SP} + O(IS_{N-1,R}R)$

C. Complexity Analysis

For HOOI algorithms, the complexity for SVD step is $O(IR^{N-1}\min(I, R^{N-1}))$. The original S^{3}TTMc costs $C^{CSS} = \sum_{l=2}^{N-1} c_{css}(l; N, R) + 2NR^{N-1}unnz$, while symmetric propagated S^{3}TTMc costs $C^{SP} = \sum_{l=2}^{N-1} c_{sp}(l; N, R) + 2NS_{N-1,R}unnz$. The maximum level l for an order-N tensor is $N-1$.

In the original HOQRI work [14], their n-ary contraction approach costs $O(R^N nnz) = O(R^N N! unnz)$, which is higher than the original S^{3}TTMc since they do not feature memoization.

In our HOQRI-SymProp, matrix multiplications on Lines 2 and 3 each have complexity $O(IS_{N-1,R}R)$, and QR decomposition on $\mathbf{A}$ costs $O(IR^2)$ which is negligible. Both our HOQRI-SymProp and HOOI-SymProp have lower time complexity than the state-of-the-art approaches.

Since HOQRI's complexity depends linearly on input dimension size I and $S_{N-1,R}$, while HOOI has a dependency on R^{N-1}. Although $S_{N-1,R}$ still grows exponentially with N and R, it grows much more slowly than R^{N-1}. This difference in complexity allows HOQRI to perform better as the input dimension size, decomposition rank and tensor order increase.

VI. Experiments

We conduct comprehensive experiments on both synthetic and real-world datasets, focusing on the performance of operations, scalability analysis, and a comparison between HOQRI and HOOI decomposition algorithms.

TABLE III: Description of sparse symmetric tensors from synthetic and real data, along with Tucker decomposition ranks.

Category	Dataset	Order	Dim	UNNZ	Rank
Synthetic	6D (L6)	6	100	10,000	2
	7D (L7)	7	400	1,000,000	3
	10D (L10)	10	400	1,000	5
	12D (H12)	12	400	10,000	3
Real	contact-school [26]	5	245	12,704	12
	trivago-clicks [27]	6	154,987	208,076	4
	walmart-trips [28]	8	62,240	47,560	10
	stackoverflow [29]	9	2,549,043	740,857	4
	amazon-reviews [30]	12	701,429	136,407	3

A. Experimental Settings

Environment. We conduct experiments on a single node of the Andes supercomputer [31] hosted by Oak Ridge Leadership Computing Facility (OLCF). The node has 256 GB of memory, with 2 AMD EPYC 7302 16-core processor at 3.0GHz (total 32 cores). Non-Uniform Memory Access (NUMA) and Sub-NUMA clustering are enabled.

Our code is written in C++ with shared memory parallelization via OpenMP and we set OMP_PROC_BIND to *spread* to ensure consistent thread scheduling. Floating-point numbers are represented in double-precision. The code is compiled with GCC 10.3.0 and linked against multithreaded OpenBLAS 3.17 [32] with OpenMP for matrix multiplication, SVD and QR decompositions.

Dataset. We evaluate our algorithm and baselines on both synthetic and real-world datasets, listed in Table III. Synthetic tensors come from the prior work [12] by requesting the authors, we choose L6, L7, L10, and H12 from their large and huge datasets. Real-world tensors are constructed from hypergraph datasets [33]. In symmetric tensor construction, each hyperedge corresponds to a unique nonzero element, with its indices representing the connected nodes. Dummy nodes are introduced to unify non-uniform hyperedge cardinalities to match the tensor order. Due to memory constraints (256 GB on our platform) and the exponential complexity of Tucker decomposition with respect to the tensor rank, we use a subset of a hypergraph by limiting the maximum hyperedge cardinality to the tensor order column in the table[1]. Real-world symmetric tensors often have large dimension sizes (i.e., the number of nodes in a hypergraph).

For synthetic datasets, we use the same ranks as in prior work [12] to enable direct comparison. For real-world datasets, the ranks are set to the maximum values that allow the baseline algorithms to run on the first two datasets, and SymProp to run within memory constraints on the remaining three datasets.

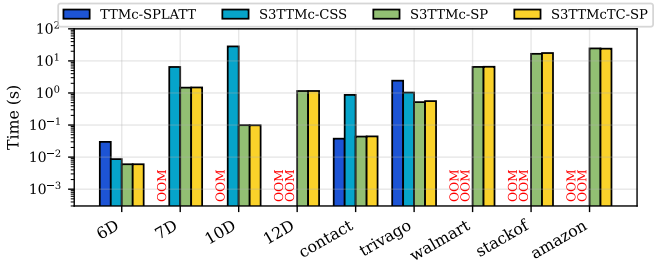


Fig. 4: Performance comparison of operations. "OOM" stands for out-of-memory.

B. Performance of Operations

We compare the performance of our approaches S^{3}TTMc-SP (*SymProp*) and S^{3}TTMcTC-SP against the baseline implementations S^{3}TTMc-CSS [11], the state-of-the-art S^{3}TTMc

[1]Except "contact-school" hypergraph is up to five-order.

implementation, and TTMc-SPLATT[2] [9], [34], the popular and efficient framework for general sparse TTMc on multicore CPUs. Each operation is run 10 times to calculate the average runtime.

1) Performance Comparison: Figure 4 compares the operation runtime in a logarithmic scale for datasets listed in Table III[3]. S^{3}TTMc-SP outperforms S^{3}TTMc-CSS by $1.44-285.49\times$, and TTMc-SPLATT by up to $4.98\times$ for the runnable cases, with the speedup increasing for higher orders and ranks. S^{3}TTMcTC-SP adds only an average of 2.1% additional runtime to S^{3}TTMc due to its extra computation. TTMc-SPLATT performs well on lower-order tensors and is the fastest on the order-5 "contact-school" dataset. However, it quickly runs out of memory for higher-order datasets. S^{3}TTMc-CSS fits larger tensors into memory, but still runs out of memory for datasets with higher orders and ranks such as "12D" and "walmart-trips".

2) Parameter Sweep: To demonstrate performance characteristics, we conduct experiments on synthetic datasets, varying one parameter at a time while keeping the others fixed. The base case is an order-7 tensor, with 10K unique non-zeros and dimension size 400 with Tucker decomposition rank 4. The plots show the average running time over 10 runs and the variance on a logarithmic scale.

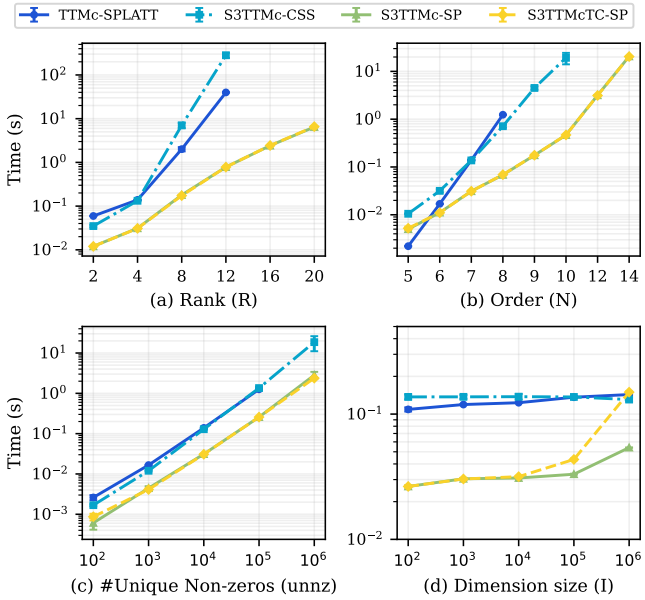


Fig. 5: Performance comparison with baselines by sweeping four parameters.

Sweep rank. Figure 5(a) shows the scalability with respect to Tucker decomposition rank. Both S^{3}TTMc-CSS and SPLATT run out of memory with Tucker rank 16 and above. S^{3}TTMc-SP achieves up to $50.9\times$ speedup over SPLATT and $360.8\times$ over S^{3}TTMc-CSS at rank 12. The runtime of S^{3}TTMc-SP increases more slowly and steadily compared to SPLATT and CSS. S^{3}TTMcTC-SP adds minimal overhead as the S^{3}TTMc step dominates its execution time.

Sweep order. Figure 5(b) demonstrates scalability with respect to input tensor order. S^{3}TTMc-SP outperforms SPLATT starting from order-6 tensors and achieves $2.10-41.2\times$ speedup over S^{3}TTMc-CSS. It successfully runs on order-14 tensors, which is 4 and 6 orders higher than S^{3}TTMc-CSS and SPLATT, respectively. The performance model of CSS [12] shows that S^{3}TTMc-CSS outperforms SPLATT for higher order and lower rank tensors, which matches the results of the cross points in Figure 5(a),(b).

Sweep number of IOUs. Figure 5(c) illustrates runtime performance with increasing IOUs. All four kernels scale linearly with the number of IOUs. Our optimized S^{3}TTMc-SP consistently outperforms SPLATT by $4.27-4.89\times$ and S^{3}TTMc-CSS by $2.75-6.58\times$. SPLATT runs out of memory for 1M number of IOUs. Since the computation added by S^{3}TTMcTC is independent of number of IOUs, the overhead of S^{3}TTMcTC-SP is more noticeable with fewer IOUs (up to 42%) but becomes negligible as the number of IOUs grows.

Sweep dimension size. Figure 5(d) shows the relationship between runtime performance and dimension size I. Although the number of floating-point operations in TTMc is independent of the dimension size, we observe a slight increase in runtime due to the increased size of the $\mathcal{Y}$ tensor, which results in slower access times and impacts performance. For S^{3}TTMcTC-SP, the times-core operation at lines 2 and 3 in Algorithm 2 scales linearly with I, contributing to the increased runtime with larger dimensions. However, this benefit would outweigh the added complexity of SVD in HOOI that will be shown in Section VI-C.

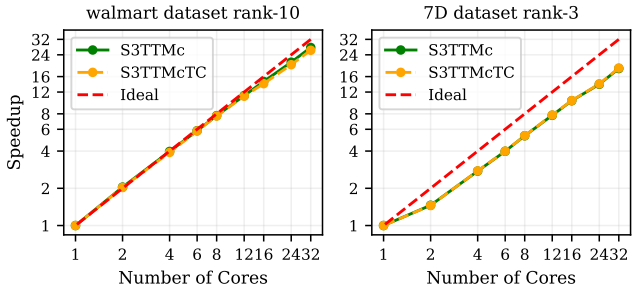


Fig. 6: Scaling of our implementation on a single Andes node

3) Thread Scalability: We analyze the thread scalability of S^{3}TTMc and S^{3}TTMcTC on the "walmart-trips" and "7D" dataset with rank 10 and 3 respectively as representatives. As shown in Figure 6, for the "walmart-trips" dataset, S^{3}TTMc and S^{3}TTMcTC achieve $27.6\times$ and $26.3\times$ speedup, respectively, on 32 cores compared to sequential execution. For the "7D" dataset, the speedups are lower at $18.6\times$ and $18.8\times$, respectively, due to less computation resulted from the lower rank.

[2]We modify the I/O in SPLATT to allow it read tensor inputs directly from IOU format to avoid reading all permutations. The MAX_MODE is set to 12 to allow it run on our benchmarked tensors in Table III.

[3]Due to the machine's memory capacity, we cannot run the "12D" dataset in the paper [12].

4) Index Iteration Analysis: We mimic the behavior of one step of the symmetric outer product in S^{3}TTMc on tensors of order 2 to 14 with decomposition ranks ranging from 3 to 8. By comparing [4] our metaprogramming approach and the index iteration method [16], our approach achieves a geometric mean speedup of $1.54\times$.

C. Behavior of Tucker Decompositions

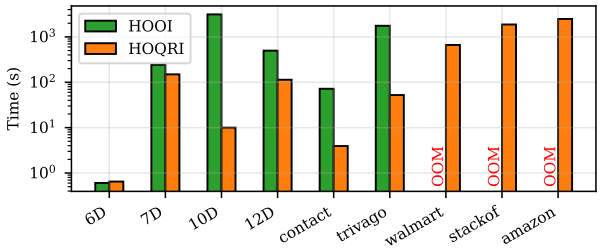


Fig. 7: Total running time comparison between HOOI and HOQRI.

1) HOQRI versus HOOI: Figure 7 shows the overall runtime of HOOI and HOQRI in our optimized approaches for 100 iterations. HOOI achieves similar or even better performance in lower-order tensors ("6D" and "7D" tensors). However, for larger ranks, higher tensor orders, or larger dimensions ("10D", "12D", and real-world tensors), HOQRI outperforms HOOI significantly, $18.3\times$ and $33.6\times$ for the "contact-school" and "trivago-clicks" datasets, respectively. This is because the SVD step limits the overall performance due to its $O(IR^{N-1}\min(I, R^{N-1}))$ complexity, which is orders of magnitude higher than the QR in HOQRI that costs merely $O(IR^2)$. HOOI runs out of memory for the last three datasets due to excessive memory requirements in the SVD step. For example, the matricized $\mathcal{Y}_{(1)}$ (of size 62K $\times$ 10M) in HOOI for the "walmart-trips" dataset requires 4.6TB of memory, while in HOQRI, $\mathcal{Y}_{psym(1)}$ (of size 62K $\times$ 11K) only occupies 5.3GB memory, resulting in a 99.88% reduction in size.

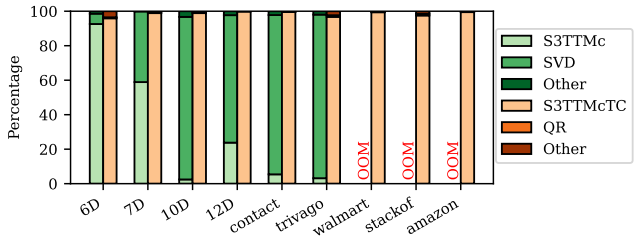


Fig. 8: Performance breakdown of HOOI and HOQRI.

2) Performance Breakdown: From the percentage breakdown in Figure 8, we notice the SVD in HOOI dominates the runtime when HOOI is much slower than HOQRI on the tensors in Figure 7. The performance gain of HOQRI over HOOI primarily comes from eliminating the SVD step of the large intermediate matrix; the S^{3}TTMcTC in HOQRI adds only a small amount of computation to S^{3}TTMc for the two matrix multiplications in Algorithm 2, as shown in Figure 5(d).

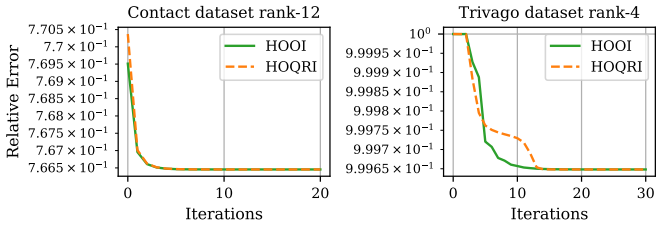


Fig. 9: Convergence comparison between HOOI and HOQRI on real-world datasets.

3) Convergence: To validate our implementation, we compare the convergence of HOOI and HOQRI on real-world datasets. Figure 9 illustrates the convergence rates of HOOI and HOQRI on the "contact-school" and "trivago-clicks" datasets as two examples. We initialize the factor matrices using HOSVD for the "contact-school" dataset and random initialization [5] for the "trivago-clicks" dataset due to the inability to run HOSVD on such a large tensor. For both datasets, the algorithms converge to the same error level, but HOOI converges faster and more stably.

VII. Related Work

Symmetric Tensor Formats. Existing works explores the format for symmetric tensors to improve computational efficiency and reduce memory usage. Ballard et al. [16] explored a compact storage format for efficient eigenvalue computation on GPUs. The Cyclops Tensor Framework (CTF) [36] extended this concept to distributed computing, using a similar compact format with padding for load balancing. Schatz et al. [15] introduced the Blocked Compact Symmetric Storage (BCSS), which uses a block structure to store IOU entries with padding, though this approach could consume more storage space for some high-order tensors. Shivakumar et al. [12] first proposed a tree-based Compressed Sparse Symmetric (CSS) format for sparse symmetric tensors. We adopted both sparse [11] and dense [16] symmetric tensor formats and applied a novel metaprogramming approach to efficiently iterate the dense symmetric tensor.

TTMc Optimizations. Various optimizations on TTMc have been proposed to enhance their efficiency. Smith et al. [9] devised an efficient TTMc algorithm on the Compressed Sparse Fiber (CSF) format [10]. Distributed computing approaches have been developed for both dense [37] and sparse [38], [39] TTMc operations. Specifically addressing symmetric tensors, Shivakumar et al. [12] optimized the S^{3}TTMc kernel based on the CSS format. Our work further

[4]Since a single run only takes microseconds, we benchmark it using Google Benchmark [35] for precise results.

[5]We randomly initialize both algorithms 20 times with different seeds and select the one with the lowest reconstruction error, following the approach in the paper [13].

optimizes S^{3}TTMc and S^{3}TTMcTC by exploiting symmetry throughout the computation process.

QR Factorization for Tensor Decomposition. Various approaches have leveraged QR factorization to improve the efficiency of tensor decomposition. The Higher-Order QR Iteration (HOQRI) [14] enhanced efficiency by replacing SVD with QR factorization. Additional research has explored different QR factorization variants, including rank-revealing QR [40] and QR with $L_{2,1}$-norm minimization [41], offering alternative approaches for tensor compression and completion. Our work adopts HOQRI due to its similarity to the HOOI algorithm in key computational routines. Additionally, our optimization ideas could inspire performance improvements in QR factorization variants when applied to sparse symmetric tensors.

Scalable Tucker Decompositions. Several algorithms have been developed to address the scalability challenges in Tucker decomposition. S-HOT [42] minimized intermediate data but used N copies of the indices data, while Singleshot [43] enabled processing of subtensors to handle larger inputs. Randomized algorithms [44]–[47] applied randomization process for reduced complexity. Distributed computation methods [38], [39], [48] leveraged multiple nodes for large-scale tensors, and GPU-accelerated algorithms [49] exploited hardware parallelism. Jin et al. [23] provided a novel scalable symmetric tensor decomposition algorithm from a mathematical perspective. Our work optimizes the TTMc kernels for both traditional HOOI and the more scalable HOQRI algorithms to improve scalability.

VIII. Conclusion

This paper introduces SymProp, a novel approach for scaling sparse symmetric Tucker decomposition via symmetry propagation. Our method significantly improves the efficiency of decomposing sparse symmetric tensors by leveraging symmetry throughout the computation process. We demonstrate substantial performance gains and enabled Tucker decompositions for higher orders, ranks and dimension sizes that previously intractable. SymProp opens new possibilities for analyzing larger-scale complex hypergraph structures and other symmetric data. While we focuse on optimizing HOOI and HOQRI algorithms, future work could explore applying the idea of propagated symmetry to other tensor decomposition methods. As the need for analyzing large-scale symmetric data continues to grow in fields such as data science and machine learning, we believe SymProp provides a significant improvement in making such analyses computationally feasible.

IX. Acknowledgement

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory and supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Contract No. DE-AC05-00OR22725. This work is also supported by the National Science Foundation under Awards No. 2247309 and 2316201 and through startup funds from the Computer Science Department at North Carolina State University.

References

[1] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009. [Online]. Available: https://doi.org/10.1137/07070111X

[2] X. Ouvrard, J.-M. L. Goff, and S. Marchand-Maillet, "Adjacency and tensor representation in general hypergraphs part 1: e-adjacency tensor uniformisation using homogeneous polynomials," 2018. [Online]. Available: https://arxiv.org/abs/1712.08189

[3] Z. T. Ke, F. Shi, and D. Xia, "Community detection for hypergraph networks via regularized tensor power iteration," *arXiv: Methodology*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:202577680

[4] M. Wang, Y. Zhen, Y. Pan, Y. Zhao, C. Zhuang, Z. Xu, R. Guo, and X. Zhao, "Tensorized hypergraph neural networks," 2024. [Online]. Available: https://arxiv.org/abs/2306.02560

[5] C. Bender, "Integral transformations. a bottleneck in molecular quantum mechanical calculations," *Journal of Computational Physics*, vol. 9, no. 3, pp. 547–554, 1972. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0021999172900101

[6] S. Sherman and T. G. Kolda, "Estimating higher-order moments using symmetric tensor decomposition," *SIAM Journal on Matrix Analysis and Applications*, vol. 41, no. 3, pp. 1369–1387, 2020. [Online]. Available: https://doi.org/10.1137/19M1299633

[7] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "Tensors for data mining and data fusion: Models, applications, and scalable algorithms," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 2, Oct. 2016. [Online]. Available: https://doi.org/10.1145/2915921

[8] L. R. Tucker, "The extension of factor analysis to three-dimensional matrices," in *Contributions to mathematical psychology.*, H. Gulliksen and N. Frederiksen, Eds. New York: Holt, Rinehart and Winston, 1964, pp. 110–127.

[9] S. Smith and G. Karypis, "Accelerating the tucker decomposition with compressed sparse tensors," in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Cham: Springer International Publishing, 2017, pp. 653–668.

[10] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 61–70.

[11] S. Shivakumar, J. Li, R. Kannan, and S. Aluru, "Efficient parallel sparse symmetric tucker decomposition for high-order tensors," in *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, 2021, pp. 193–204.

[12] ——, "Sparse symmetric format for tucker decomposition," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 6, pp. 1743–1756, 2023.

[13] L. De Lathauwer, B. De Moor, and J. Vandewalle, "On the best rank-1 and rank-(r1 ,r2 ,. . .,rn) approximation of higher-order tensors," *SIAM Journal on Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1324–1342, 2000. [Online]. Available: https://doi.org/10.1137/S0895479898346995

[14] Y. Sun and K. Huang, "Hoqri: Higher-order qr iteration for scalable tucker decomposition," in *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2022, pp. 3648–3652.

[15] M. D. Schatz, T. M. Low, R. A. van de Geijn, and T. G. Kolda, "Exploiting symmetry in tensors for high performance: Multiplication with symmetric tensors," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C453–C479, 2014. [Online]. Available: https://doi.org/10.1137/130907215

[16] G. Ballard, T. Kolda, and T. Plantenga, "Efficiently computing tensor eigenvalues on a gpu," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 1340–1348.

[17] J. Li, J. Sun, and R. Vuduc, "Hicoo: Hierarchical storage of sparse tensors," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 238–252.

[18] A. E. Helal, J. Laukemann, F. Checconi, J. J. Tithi, T. Ranadive, F. Petrini, and J. Choi, "Alto: adaptive linearized storage of sparse tensors," in *Proceedings of the 35th ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 404–416. [Online]. Available: https://doi.org/10.1145/3447818.3461703

[19] I. Nisa, J. Li, A. Sukumaran-Rajam, P. S. Rawat, S. Krishnamoorthy, and P. Sadayappan, "An efficient mixed-mode representation of sparse tensors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356216

[20] TensorNetwork.org, "Tensor network diagram notation," https://tensornetwork.org/diagrams/, accessed: 2024-10-02.

[21] M. Lysenko, "symmetric-tensor-index," 2024. [Online]. Available: https://github.com/mikolalysenko/symmetric-tensor-index

[22] A. Abbott, "Symtensor," 2024. [Online]. Available: https://github.com/adabbott/SymTensor

[23] R. Jin, J. Kileel, T. G. Kolda, and R. Ward, "Scalable symmetric tucker tensor decomposition," *SIAM Journal on Matrix Analysis and Applications*, vol. 45, no. 4, pp. 1746–1781, 2024. [Online]. Available: https://doi.org/10.1137/23M1582928

[24] L. De Lathauwer, B. De Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM Journal on Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1253–1278, 2000. [Online]. Available: https://doi.org/10.1137/S0895479896305696

[25] P. A. Regalia, "Monotonically convergent algorithms for symmetric tensor approximation," *Linear Algebra and its Applications*, vol. 438, no. 2, pp. 875–890, 2013, tensors and Multilinear Algebra. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0024379511007300

[26] J. Stehlé, N. Voirin, A. Barrat, C. Cattuto, L. Isella, J.-F. Pinton, M. Quaggiotto, W. V. den Broeck, C. Régis, B. Lina, and P. Vanhems, "High-resolution measurements of face-to-face contact patterns in a primary school," *PLoS ONE*, vol. 6, no. 8, p. e23176, 2011. [Online]. Available: https://doi.org/10.1371/journal.pone.0023176

[27] P. S. Chodrow, N. Veldt, and A. R. Benson, "Hypergraph clustering: from blockmodels to modularity," *Science Advances*, 2021.

[28] I. Amburg, N. Veldt, and A. R. Benson, "Clustering in graphs and hypergraphs with categorical edge labels," in *Proceedings of the Web Conference*, 2020.

[29] N. Veldt, A. R. Benson, and J. Kleinberg, "Minimizing localized ratio cut objectives in hypergraphs," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2020.

[30] J. Ni, J. Li, and J. McAuley, "Justifying recommendations using distantly-labeled reviews and fine-grained aspects," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 188–197.

[31] Oak Ridge Leadership Computing Facility, "Andes user guide," 2024, accessed: 2024-10-06. [Online]. Available: https://docs.olcf.ornl.gov/systems/andes_user_guide.html

[32] Z. Xianyi, M. Kroeker, and OpenBLAS Project, "Openblas," 2024. [Online]. Available: http://www.openblas.net/

[33] A. R. Benson, "Austin R. Benson datasets," https://www.cs.cornell.edu/~arb/data/, accessed: 2024-10-03.

[34] S. Smith and G. Karypis, "SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit," http://cs.umn.edu/ splatt/, 2016.

[35] Google, "Google benchmark," 2024. [Online]. Available: https://github.com/google/benchmark

[36] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 813–824.

[37] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, X. Liu, P. Murali, Y. Sabharwal, and D. Sreedhar, "On optimizing distributed tucker decomposition for dense tensors," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1038–1047.

[38] O. Kaya and B. Uçar, "High performance parallel algorithms for the tucker decomposition of sparse tensors," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 103–112.

[39] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, P. Murali, S. S. Pandian, Y. Sabharwal, and D. Sreedhar, "On optimizing distributed tucker decomposition for sparse tensors," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 374–384. [Online]. Available: https://doi.org/10.1145/3205289.3205315

[40] M. Beaupère, D. Frenkiel, and L. Grigori, "Higher-order qr with tournament pivoting for tensor compression," *SIAM Journal on Matrix Analysis and Applications*, vol. 44, no. 1, pp. 106–127, 2023. [Online]. Available: https://doi.org/10.1137/20M1387663

[41] Y. Zheng and A.-B. Xu, "Tensor completion via tensor qr decomposition and l2,1-norm minimization," *Signal Processing*, vol. 189, p. 108240, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0165168421002772

[42] J. Oh, K. Shin, E. E. Papalexakis, C. Faloutsos, and H. Yu, "S-hot: Scalable high-order tucker decomposition," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, ser. WSDM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 761–770. [Online]. Available: https://doi.org/10.1145/3018661.3018721

[43] A. Traoré, M. Berar, and A. Rakotomamonjy, "Singleshot: a scalable tucker tensor decomposition," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[44] P. Drineas and M. W. Mahoney, "A randomized algorithm for a tensor-based generalization of the singular value decomposition," *Linear Algebra and its Applications*, vol. 420, no. 2, pp. 553–571, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0024379506003867

[45] R. Minster, A. K. Saibaba, and M. E. Kilmer, "Randomized algorithms for low-rank tensor decompositions in the tucker format," *SIAM Journal on Mathematics of Data Science*, vol. 2, no. 1, pp. 189–215, 2020. [Online]. Available: https://doi.org/10.1137/19M1261043

[46] M. Che and Y. Wei, "Randomized algorithms for the approximations of tucker and the tensor train decompositions," *Advances in Computational Mathematics*, vol. 45, no. 1, pp. 395–428, 2019.

[47] S. Ahmadi-Asl, S. Abukhovich, M. G. Asante-Mensah, A. Cichocki, A. H. Phan, T. Tanaka, and I. Oseledets, "Randomized algorithms for computation of tucker decomposition and higher order svd (hosvd)," *IEEE Access*, vol. 9, pp. 28 684–28 706, 2021.

[48] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 1047–1058.

[49] J. Lee, D. Han, O.-K. Kwon, K.-W. Chon, and M.-S. Kim, "Gputucker: Large-scale gpu-based tucker decomposition using tensor partitioning," *Expert Systems with Applications*, vol. 237, p. 121445, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417423019474