Efficient Parallel Sparse Symmetric Tucker Decomposition for High-Order Tensors

Shruti Shivakumar*, Jiajia Li†, Ramakrishnan Kannan‡, Srinivas Aluru§

Abstract

Tensor based methods are receiving renewed attention in recent years due to their prevalence in diverse real-world applications. There is considerable literature on tensor representations and algorithms for tensor decompositions, both for dense and sparse tensors. Many applications in hypergraph analytics, machine learning, psychometry, and signal processing result in tensors that are both sparse and symmetric, making it an important class for further study. Similar to the critical Tensor Times Matrix chain operation (TTMc) in general sparse tensors, the Sparse Symmetric Tensor Times Same Matrix chain (S^3TTMc) operation is compute and memory intensive due to high tensor order and the associated factorial explosion in the number of non-zeros. In this work, we present a novel compressed storage format CSS for sparse symmetric tensors, along with an efficient parallel algorithm for the S^3TTMc operation. We theoretically establish that S^3TTMc on CSS achieves a better memory versus run-time trade-off compared to state-of-the-art implementations. We demonstrate experimental findings that confirm these results and achieve up to 2.9× speedup on synthetic and real datasets.

1 Introduction

Tensors are higher dimension generalizations of matrices, and are used to represent multi-dimensional data. Symmetric tensors are an important class of tensors, arising in diverse fields such as psychometry, signal processing, machine learning, and hypergraph analytics [42, 31, 2, 14, 12, 18]. Estimating means of Gaussian graphical models and independent component analysis often utilize symmetric tensors for decompositions [4, 15, 2]. Hypergraphs, generalizations of graphs which allow edges to span multiple vertices, have become ubiquitous in understanding real world networks and multi-entity interactions [10, 11]. Affinity relations in a hypergraph can be represented as a high-order adjacency tensor which is sparse and symmetric [13, 29, 2, 44]. While mathematical research on symmetric tensors is longstanding [30, 20, 7, 9], emerging massive data in these applications has sparked the demand for scalable, efficient algorithms that utilize advances in numerical linear algebra, numerical optimization, as well as high performance computing. State-of-the-art tensor libraries [35, 21, 40] incorporate high performance tensor methods for general sparse tensors; however, to the best of our knowledge, they lack specialized algorithms for sparse tensors that are symmetric.

A representative tensor approach in dimensionality reduction and hypergraph clustering is tensor decomposition, and the popular Tucker decomposition [19] is studied in this work. A key kernel in symmetric Tucker decomposition is the *Sparse Symmetric Tensor Times Same Matrix chain* (S^3TTMc) operation[1], a specialization of the fundamental tensor times matrix chain operation (TTMc) for symmetric tensors. In symmetric Tucker decomposition, instead of different matrix factors in the chain, the same matrix is repeatedly used.

In high-performance algorithms for sparse tensor decomposition, a balance between two desirable but competing goals is pursued - (i) compressed storage format to represent the sparse tensor, and (ii) efficient computation based on it to perform the decomposition. For general sparse tensors, the two goals could align well [35, 22, 24, 27] since when a sparse tensor is stored compactly, the memory footprint of the tensor - which could be a big portion of the memory traffic during computation - is reduced. Moreover, superior sparse tensor compression can co-exist with better data locality. For example, a compact Compressed Sparse Row (CSR) representation of a sparse matrix demonstrates better data locality from the reuse of row indices. A similar phenomenon occurs in Compressed Sparse Fibers (CSF)

*Georgia Institute of Technology, Atlanta, GA, USA

†Pacific Northwest National Laboratory, Richland, WA, USA; William & Mary, Williamsburg, VA, USA. This research was also partially funded by the US Department of Energy under Award No. 66150 and the Laboratory Directed Research and Development program at PNNL under contract No. ND8577.

‡Oak Ridge National Laboratory, Oak Ridge, TN, USA. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (`http://energy.gov/downloads/doe-public-access-plan`).

§Georgia Institute of Technology, Atlanta, GA, USA

[1]For notational convenience, we call S^3TTMc in lieu of SSTTSMc.

[35] and Hierarchical COOrdinate (HiCOO) [22] formats from different perspectives.

For sparse symmetric tensors, efficient computation and compressed storage format are *contending characteristics*, especially for high dimensional data. Fig. 1 illustrates the trade-off between efficiently computing S^{3}TTMc and the size of compressed storage data structures for symmetric tensors. The *Unique COOrdinate* (*UCOO*) representation achieves compact storage by saving the coordinates (indices) along with the value of each non-zero exactly once. However, the compression comes at the price of slow S^{3}TTMc computation, as retrieving all index permutations of every non-zero increases redundant computation and limits parallelism. On the other hand, by saving all index permutations of each non-zero, we can overcome the computation challenges in *UCOO*. Thus, S^{3}TTMc becomes a general TTMc where state-of-the-art high performance libraries, such as SPLATT [35] and ParTI! [21], can be leveraged. Unfortunately, this approach leads to $\mathcal{O}(N!)$ increase in storage, where N is the tensor order. With the pervasiveness of high order adjacency tensors representing massive hypergraphs, the memory overhead is formidable. For example, an order-14 sparse symmetric tensor generated from Amazon reviews [26] (Details in Sec. 6.2) causes $14! = 8.7 \times 10^{10}$ factor increase in storage to save all index permutations. Thus, one is forced to choose between efficient algorithms that use storage formats that cannot be sustained for high-order tensors (*SPLATT*), and efficient storage formats (*UCOO*) that do not permit efficient algorithms.

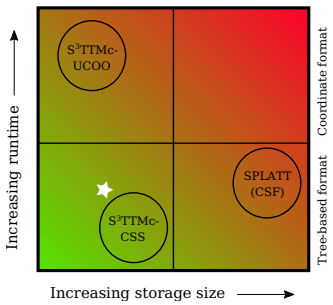


Figure 1: Comparison of trade-off between run-time and storage size for proposed S^{3}TTMc-CSS computation to different algorithms and storage formats.

In this paper, we propose a *computation-aware compact storage format*, termed the Compressed Sparse Symmetric (CSS) format, which has storage requirement of the same order as *UCOO* while being able to perform S^{3}TTMc more efficiently than state-of-the-art TTMc implementations, such as SPLATT, as shown in Fig. 1. CSS is especially suited for high-order tensors since its size is bounded by 2^N, asymptotically smaller than $N!$. We present a novel shared-memory parallel algorithm S^{3}TTMc-CSS to efficiently perform S^{3}TTMc using CSS. We demonstrate through experiments that the performance of our parallel algorithm concurs with Fig. 1.

Our main contributions are summarized as follows:

- We propose the novel computation-aware CSS structure for sparse symmetric tensors. (Sec. 4)
- We design and implement an efficient multi-core parallel S^{3}TTMc algorithm, called S^{3}TTMc-CSS based on the CSS format. (Sec. 5)
- We perform a thorough cost analysis of S^{3}TTMc-CSS and outline a detailed comparison to baseline implementations and their formats. (Secs. 4.2 and 5.5)
- Our S^{3}TTMc-CSS bridges the gap between efficient S^{3}TTMc computation and compact storage - S^{3}TTMc-CSS is (i) up to 2.9× faster than SPLATT while requiring up to five orders of magnitude lesser memory, and (ii) at least two orders of magnitude faster than *S^{3}TTMc-UCOO* while only requiring up to 6× memory than *UCOO*. (Sec. 6)
- We report, for the first time, the Tucker decomposition of an order-14 tensor from a real world hypergraph. (Sec. 6.6)

2 Background

2.1 Sparse Symmetric Tensors An order-N symmetric tensor $\mathcal{X} \in \mathbb{R}^{I \times I \times \ldots I}$ can be generated from an N-uniform hypergraph on I vertices, where every hyperedge spanning multiple vertices has the same cardinality N. The tensor order N is the number of modes or dimensions. The non-zero values of the symmetric tensor $\mathcal{X}$ remain unchanged under any permutation of its indices. That is, for every $e = \{v_{i_1}, v_{i_2}, \ldots, v_{i_N}\} \in E$, $\mathcal{X}_{\sigma(\mathbf{i})} = w(e)$, where $\sigma(\mathbf{i})$ denotes any of the $N!$ permutations of the index tuple $\mathbf{i} = (i_1, \ldots, i_N)$. We refer to a non-zero $\mathcal{X}_{\mathbf{i}} \in unz(\mathcal{X})$ as an *index-ordered unique (IOU) non-zero* if the index set $\mathbf{i}$ is ordered $(i_1 < i_2 < \ldots < i_N)$. Thus, $nz(\mathcal{X})$ is divided into a set of equivalence classes, $unz(\mathcal{X})$ of size $unnz \ll nnz$, under the permutation relation σ. The IOU non-zero $\mathcal{X}_{\mathbf{i}}$ represents an equivalence class containing all permutations $\sigma(\mathbf{i})$ of the index set $\mathbf{i}$. A subtensor of a tensor $\mathcal{X}$ is obtained by fixing indices along some dimensions while varying along the others, denoted by ':'. The matricization of $\mathcal{X}$ along any mode n $(1 \leq n \leq N)$ flattens/unfolds the tensor into a matrix $\mathbf{X}_{(n)} \in \mathbb{R}^{I \times I^{N-1}}$. Following Kolda and Bader [19], we denote vectors using bold lowercase letter (e.g., $\mathbf{i}$, $\mathbf{j}$), matrices using bold uppercase letters (e.g., $\mathbf{U}$), and tensors using bold calligraphic letters (e.g., $\mathcal{X}$).

2.2 Sparse Tensor Formats Multiple compressed data structures, with varying levels of compactness, have been proposed for general sparse tensors. Popular among these include COO [19], CSF [35], MM-CSF [27],

F-COO [24], and HiCOO [23] formats. The simplest storage format for sparse tensors is the COOrdinate (COO) format. COO stores each non-zero as a tuple $(i_1, i_2, \ldots i_N; v)$, where $i_n, n = 1, \ldots, N$ is an index coordinate and v is the non-zero value. The Compressed Sparse Fiber (CSF) format is a higher order analogue to the Compressed Sparse Row (CSR) sparse matrix format. It is a compressed storage format structured as a tree, where every level corresponds to a tensor mode. Every non-zero in a sparse tensor is represented by a root to leaf path.

Algorithm 1: Symmetric HOOI

Result: Core tensor $\mathcal{C}$, and orthonormal matrix $\mathbf{U}$

1 **while** $\mathcal{C}$ *not converged* **do**
2 $\quad \mathcal{Y} = \mathcal{X} \times_{-1} [\mathbf{U}]$ // S^{3}TTMc
3 $\quad \mathbf{U} \leftarrow R$ left singular vectors of matricized $\mathcal{Y}$
4 $\quad \mathcal{C} = \mathcal{Y} \times_1 \mathbf{U}$

2.3 Sparse Symmetric Tucker Decomposition

Symmetric Tucker decomposition of an order-N sparse symmetric tensor $\mathcal{X} \in \mathbb{R}^{I \times I \ldots I}$ finds a dense orthonormal matrix $\mathbf{U} \in \mathbb{R}^{I \times R}$ and an order-N dense symmetric core tensor $\mathcal{C} \in \mathbb{R}^{R \times R \ldots R}$ such that $\arg\min_{\mathcal{C},\mathbf{U}} ||\mathcal{X} - \mathcal{C} \times_1 \mathbf{U} \times_2 \mathbf{U} \ldots \times_N \mathbf{U}||$. In hypergraph clustering and dimensionality reduction, the matrix rank R corresponds to number of clusters and size of reduced space respectively, and is usually a small value. We show the symmetric higher order iterations (HOOI) algorithm (Algorithm 1) by introducing the symmetry property into the popular HOOI approach to Tucker decomposition [19]. The operation in Line 2, *Sparse Symmetric Tensor Times Same Matrix chain* (S^{3}TTMc), is observed to be computationally expensive in large data, and will be our focus.

$$\mathcal{Y} = \mathcal{X} \times_{-1} [\mathbf{U}] = \mathcal{X} \times_2 \mathbf{U} \times_3 \mathbf{U} \ldots \times_N \mathbf{U} \tag{2.1}$$

$$\implies \mathbf{Y}(i_1, :) = \sum_{\mathcal{X}_\mathbf{i} \in nz(\mathcal{X})} \mathcal{X}_\mathbf{i} \left\{ \bigotimes_{j=2}^{N} \mathbf{U}(i_j, :) \right\} \tag{2.2}$$

$$= \sum_{\mathcal{X}_{i_1,:,\ldots}} \mathbf{U}(i_2, :) \otimes \ldots \left(\sum_{\mathcal{X}_{\ldots i_{N-1},:}} \mathcal{X}_{\ldots i_{N-1},:} \mathbf{U}(i_N, :) \right) \tag{2.3}$$

S^{3}TTMc, given by Eq. (2.1), is represented by a sequence of tensor times matrix products (TTM) [34] on all but one mode of the symmetric tensor $\mathcal{X}$. TTM of an order-N symmetric tensor $\mathcal{X} \in \mathbb{R}^{I \times I \ldots I}$ with a dense matrix $\mathbf{U} \in \mathbb{R}^{I \times R}$ along mode n, denoted by $\mathcal{Y} = \mathcal{X} \times_n \mathbf{U}$, is defined for $r \in \{1, \ldots, R\}$ as $\mathcal{Y}_{i_1 \ldots i_{n-1} r i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I} \mathcal{X}_{i_1 \ldots i_{n-1} i_n i_{n+1} \ldots i_N} \mathbf{U}_{i_n r}$.

The Kronecker product of vectors $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$, denoted by $\mathbf{u} \otimes \mathbf{v} \in \mathbb{R}^{mn}$, is the vectorized outer product [19]. The sparsity of $\mathcal{X}$ favors rewriting S^{3}TTMc using Kronecker products in Eq. (2.2), similar to the approach used in the work [16, 32], by only doing meaningful computation corresponding to its non-zeros (with index permutations), thus obtaining the matricized output, $\mathbf{Y}$. Eq. (2.3) also utilizes the distributivity of Kronecker products to reuse intermediate results across multiple non-zeros, named *non-zero memoization*.

2.4 Other Related Work

Sparse Tensor Decompositions State-of-the-art sparse tensor CANDECOMP/PARAFAC (CP) decomposition [36, 22, 27] and Tucker decomposition [25, 34, 32] research targets general sparse tensors. Kaya *et al.* proposed a dimension tree data structure that partitions mode indices in a hierarchical manner [17] to efficiently utilize intermediate results among TTM operations, so-called *operation memoization*, thus reducing the cost of CP decomposition. The *operation memoization* [17], orthogonal with our *nonzero memoization*, could be utilized to further enhance S^{3}TTMc performance in the future.

Dense Symmetric Tensor Decompositions There are optimized approaches designed for dense symmetric tensor methods [8]. Ballard *et al.* [5] designed a symmetric storage format for dense tensor eigenvalue algorithms on GPUs. Schatz *et al.* [28] proposed a blocked compact symmetric storage to reduce computation redundancy by utilizing temporary tensors. Solomonik *et al.* proposed symmetric tensor contractions with fewer multiplications [37] and proved lower bounds for communication [38]. Cai *et al.* [6] explored symmetry to reduce redundancy in computation and storage in INDSCAL (individual differences in scaling) decomposition. To the best of our knowledge, the work presented in this paper is the first to effectively utilize symmetry in sparse tensors for space- and performance-efficiency.

3 Baseline Implementations

We use three existing implementations based on two general sparse tensor formats as our baselines – *UCOO*, the 'unique COO' format that stores only IOU non-zeros, and *FCSF*, the 'full CSF' format which stores all index permutations. Two S^{3}TTMc implementations using *UCOO* are *S^{3}TTMc-UCOO* and Cyclops Tensor Framework (CTF) [40, 39]; we use *SPLATT* [33] to evaluate S^{3}TTMc implementation using *FCSF*.

$$\mathbf{Y}(ind, :) = \sum_{\substack{\mathcal{X}_\mathbf{i} \in unz(\mathcal{X}) \\ ind \in \mathbf{i}}} \mathcal{X}_\mathbf{i} \sum_{\mathbf{j} = \sigma(\mathbf{i} \setminus ind)} \left\{ \bigotimes_{k=2}^{N} \mathbf{U}(j_k, :) \right\} \tag{3.4}$$

3.1 *S^{3}TTMc-UCOO and CTF*

Among the state-of-the-art formats, COO directly supports storing only IOU non-zeros of a sparse symmetric tensor because of its invariance with mode order [23]. Note that

in Eq. (2.2), an IOU non-zero $\mathcal{X}_{i_1,\ldots i_N}$ contributes $(N-1)!$ terms to the summation for each of the rows $\mathbf{Y}(i_1,:), \mathbf{Y}(i_2,:), \ldots, \mathbf{Y}(i_N,:)$, i.e., every permutation $(j_1,\ldots j_N) = \sigma(i_1, i_2, \ldots i_N)$ contributes a term $(\mathbf{U}(j_2,:) \otimes \ldots \otimes \mathbf{U}(j_N,:))$ to the summation in Eq. (2.2) for $\mathbf{Y}(j_1,:)$. We implement a parallel *S³TTMc-UCOO* baseline by grouping terms in Eq. (2.2) and accessing only the IOU non-zeros $unz(\mathcal{X})$ for the summation, as illustrated in Eq. (3.4). However, computing S³TTMc on *UCOO* is inefficient since updating the rows of $\mathbf{Y}$ in parallel raises memory contention, thus requiring atomic operations as all IOU non-zeros whose indices contain i_n contribute to $\mathbf{Y}(i_n,:)$. We evaluate the performance of our implementation of Eq. (3.4) and the existing Cyclops Tensor Framework (CTF) [40], both using the *UCOO* format, in computing S³TTMc in *S³TTMc-UCOO* and *CTF* baselines respectively. A disadvantage of both baselines is that they cannot perform *non-zero memoization*, i.e. memoizing intermediate permutation results among IOU non-zeros, without maintaining additional information on index distribution.

3.2 *SPLATT using FCSF format* The CSF format is more storage-efficient than COO but suffers from enforcing mode ordering. If we only store IOU non-zeros in a CSF representation, in order to update $\mathbf{Y}(i_n,:)$ for a given IOU non-zero $\mathcal{X}_{\mathbf{i}}$, traversing the path corresponding to $\mathcal{X}_{\mathbf{i}}$ $\mathcal{O}(N!)$ times is unavoidable. This is because CSF only considers non-zero memoization among IOU non-zeros, but not within the permutations of an IOU non-zero. Thus, by assuming a machine with sufficiently large memory, we store all index permutations of IOU non-zeros in CSF format and use the state-of-the-art TTMc algorithm for general sparse tensors - SPLATT [35] - as our third baseline to compute S³TTMc.

4 CSS Structure

Our objective is to design a data structure for sparse symmetric tensors which reduces memory consumption and avoids redundant saving of symmetric information, overcoming one of the key challenges affecting the *FCSF* format. Moreover, the format must enable two types of non-zero memoization for intermediate Kronecker product results in the S³TTMc operation - (i) between IOU non-zeros, stored in a tree structure similar to the CSF format, and (ii) within permutations of an IOU non-zero, uniquely designed for symmetry in this work. We trade off memory requirement for efficient S³TTMc computation, thereby outperforming the baselines using *UCOO* representation.

Inspired by the CSF format, we design a tree structure that stores only IOU non-zeros. We name our tree data structure as the *Compressed Sparse Symmetric* (CSS) format. Consider a symmetric tensor $\mathcal{X} \in \mathbb{R}^{I\times I\times\ldots I}$ of order N. CSS is structured as a forest with $N-1$ levels constructed from IOU non-zeros, $unz(\mathcal{X})$. We denote the path down the CSS tree from level 1 to node i_l at level l by $\mathcal{P}_{\mathbf{i}_l} = (i_1, i_2, \ldots i_l)$. Then, $\mathcal{P}_{\mathbf{i}_l}$ corresponds to an ordered subsequence of size l contained in at least one ordered index tuple $\mathbf{j}$, i.e. $\mathcal{P}_{\mathbf{i}_l} = (j_{k_1}, j_{k_2}, \ldots j_{k_l}) \subseteq \mathbf{j}$ while maintaining the order $k_1 < \ldots < k_l$. Storing all ordered subsequences of IOU non-zeros highlights the computation-aware property of the CSS format - both forms of non-zero memoization are achieved by our S³TTMc operation with minimal additional index information. In the example shown in Fig. 2, the CSS format constructed from the 6 non-zeros of the order-4 tensor in Table 1 has 3 levels. Nodes $(2,3,5)$ and $(2,8)$ form two paths of the non-zero $(2,3,5,8)$. Due to the lexicographical order, the path $(2,8)$ has to stop at the maximum index 8, thus its length is only 2. Each leaf node at level $(N-1)$ stores an array containing the left-out index $i_N = \mathbf{i} \backslash \mathcal{P}_{\mathbf{i}_{N-1}}$ and the non-zero value (shown in blue in Fig. 2 and Table 1). In the example, the array $[8, 1.0]$ is affiliated to node-5 in path $\mathcal{P}_{\mathbf{5}} = (2,3,5)$. The leaf nodes in levels higher than $N-1$ are essential to efficiently computing S³TTMc (Details in Sec. 5).

Note that the paths $\mathcal{P}_{\mathbf{i}_l}$ correspond to l-length subsequences of index tuples, while $\mathbf{i}$ refer to the index tuples themselves. Thus, each IOU non-zero corresponds to multiple paths $\mathcal{P}_{\mathbf{i}_l}$ to levels $l = 1, 2, \ldots N-1$. In Fig. 2, we mark the nodes on paths constructed for the first non-zero $(2,3,5,8)$ in red. There are six paths of length 1 from level 1 to level 2, and four paths of length 2 from level 1 to level 3; thus fourteen paths in total of lengths 0, 1 and 2. Generally, each IOU non-zero contributes $\binom{N}{l}$ nodes to level l and totally (2^N-1) nodes to the CSS representation. Across multiple non-zeros, their paths/nodes could be reused, e.g., path $(2,3)$ belongs to both $(2,3,5,8)$ and $(1,2,3,9)$ non-zeros. Thus, the total number of nodes in CSS tree depends on the IOU non-zero distribution. We empirically observe that for a uniform distribution of IOU non-zeros in the tensor, the number of nodes in CSS is much smaller than $(2^N-1)unnz$ due to the overlap of nodes between non-zeros. Such a construction not only provides a compact representation of a symmetric tensor, but also ensures uniqueness in representation. We present two key properties of the CSS data structure that are useful in the construction of the S³TTMc-CSS algorithm below.

PROPERTY 4.1. *Any path in the CSS forest exists only if it is a subsequence of the index tuple of at least one IOU non-zero. Moreover, the ordering of the node values in every path further ensures unique representation of permutations of subsets of all IOU non-zeros.*

Due to Property 4.1, there are exactly $N = \binom{N}{N-1}$

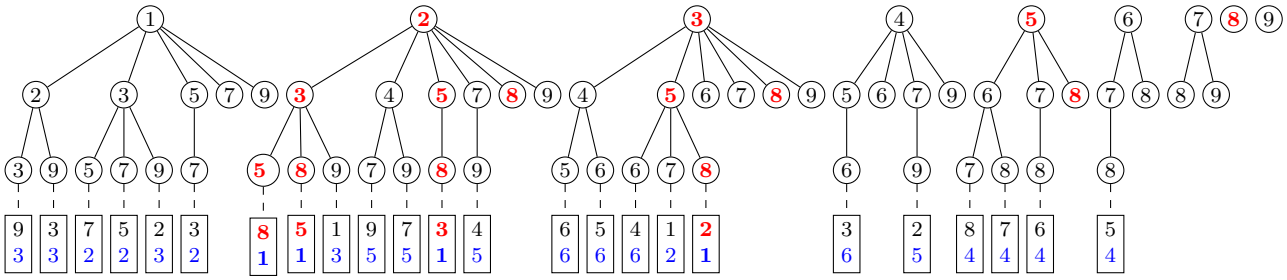


Figure 2: CSS format for the sparse symmetric tensor in Table 1

i_1	**2**	1	1	5	2	3
i_2	**3**	3	2	6	4	4
i_3	**5**	5	3	7	7	5
i_4	**8**	7	9	8	9	6
vals	**1**	2	3	4	5	6

Table 1: A sparse symmetric tensor $\mathcal{X} \in \mathbb{R}^{10\times10\times10\times10}$ with 6 IOU nonzeros.

distinct paths to the leaf nodes at level $(N-1)$ for any IOU non-zero. In the example in Fig. 2, four paths repeatedly save the index tuple $(2,3,5,8)$ at level-3 with values.

PROPERTY 4.2. *There exists an injective mapping between a set of IOU non-zeros of an order-N sparse symmetric tensor and the CSS forest constructed from it.*

While the CSS format still has some redundancy in terms of storing subsequences of index tuples of IOU non-zeros, it is a necessity for efficient S^{3}TTMc computation (discussed in Sec. 5). Moreover, CSS is practical, as it is less expensive than the state-of-the-art general sparse formats (compared below).

4.1 Space Complexity For a single IOU non-zero, the number of nodes at level l is $\binom{N}{l}$ for CSS. We need to save both pointers and indices for every level, which doubles the storage of CSS. The affiliated leaf level indices and non-zero values are stored in each of the N nodes at level $(N-1)$ generated by this non-zero. Thus, the space requirements for storing one IOU non-zero unz, and for storing $unnz$ non-zeros are given by Eq. (4.5) and Eq. (4.6) respectively.

$$S^{CSS}_{unz} = 2\beta_{int}\sum_{l=1}^{N-1}\binom{N}{l} + N(\beta_{int}+\beta_{dbl}) \tag{4.5}$$

$$S^{CSS}_{total} = 2\beta_{int}\sum_{l=1}^{N-1} nnz_l + unnz\cdot N(\beta_{int}+\beta_{dbl}) \tag{4.6}$$

where nnz_l is the number of nodes at level l in the CSS structure. The overlap in the nodes across IOU non-zeros is inherent in the nnz_l parameter, thus $nnz_l < \binom{N}{l}\cdot unnz$. However, as the CSS structure has a combinatorial construction procedure, the sizes of the levels follow the trend of the binomial coefficient, $nnz_1 < nnz_2 \ldots < nnz_{N/2} > \ldots > nnz_{N-1}$. We use a binary search tree for inserting nodes in CSS in order to maintain the ordering of nodes while traversing depth-wise and level-wise through the CSS tree. Thus, the time complexity of CSS construction is $\mathcal{O}\big(unnz\log I\big(2^N-N\big)\big)$.

4.2 Comparison to baselines

UCOO: Storing IOU non-zeros in *UCOO* is more compact than CSS. However, as we will demonstrate in Sec. 6, S^{3}TTMc using the CSS format is more work efficient than using *UCOO*.

$$S^{UCOO}_{total} = (N\beta_{int}+\beta_{dbl})unnz \tag{4.7}$$

FCSF: Though CSS and *FCSF* both use the tree structure to explore overlap between non-zeros, unlike CSS, *FCSF* does not recognize the symmetry feature of a tensor, and has $N!\cdot unnz$ root to leaf paths. We give the space analysis for *FCSF* by referring to the work [34]. For a single IOU non-zero unz, each *FCSF* level has $\binom{N}{l}l!$ nodes for all permutations, consuming a factorial order more space than CSS (Eq. (4.8)). The storage overhead of $nnz = unnz\cdot N!$ non-zeros in $\mathcal{X}$ is given by Eq. (4.9).

$$S^{FCSF}_{unz} = 2\beta_{int}\sum_{l=1}^{N-1}\binom{N}{l}l! + N!\,(\beta_{int}+\beta_{dbl}) \tag{4.8}$$

$$S^{FCSF}_{total} = 2\beta_{int}\sum_{l=1}^{N-1} nnz'_l + N!\,unnz(\beta_{int}+\beta_{dbl}) \tag{4.9}$$

where nnz'_l is the number of nodes at level l in the CSF format with $nnz'_1 < nnz'_2 < \ldots < nnz'_{N-1}$. Though the last term in Eq. (4.6) is much smaller than Eq. (4.9), it is still difficult to directly compare them due to the significant overlapping among nodes reflected in nnz_l and nnz'_l respectively. We will experimentally validate our analysis so far by demonstrating the space advantages of CSS on synthetic and real data in Sec. 6.

5 S^{3}TTMc Computation

In this section, we propose an algorithm to compute S^{3}TTMc for sparse symmetric tensors stored in CSS format. We seek to remove the arithmetic redundancies in the implementation of Eq. (3.4) for *S^{3}TTMc-UCOO* baseline, and demonstrate how CSS inherently supports a memoization strategy that achieves this goal.

5.1 Formulation We recursively define vector function $\mathcal{K}$ which allows for further factorization of the Kronecker products.

$$\mathcal{K}(\mathbf{i}) = \begin{cases} \mathbf{U}(i,:) & |\mathbf{i}| = 1 \\ \sum_{j \in i_1 \dots i_l} \mathbf{U}(j,:) \otimes \mathcal{K}(\mathbf{i} \setminus j) & |\mathbf{i}| = l > 1 \end{cases} \tag{5.10}$$

If we unroll the recursion in Eq. (5.10), note that $\mathcal{K}(\mathbf{i})$ is the sum of Kronecker products of rows of $\mathbf{U}$ for every permutation $\mathbf{j} = \sigma(\mathbf{i})$ (Eq. (5.11)). Then, Eq. (3.4) can be succinctly factorized using Eq. (5.10), as seen in Eq. (5.12).

$$\mathcal{K}(\mathbf{i}) = \sum_{\mathbf{j}=\sigma(\mathbf{i})} \otimes_k \mathbf{U}(j_k,:) \tag{5.11}$$

$$\mathbf{Y}(ind,:) = \sum_{\substack{\mathcal{X}_\mathbf{i} \in unz(\mathcal{X}) \\ ind \in \mathbf{i}}} \mathcal{X}_\mathbf{i} \, \mathcal{K}(\mathbf{i} \setminus ind) \tag{5.12}$$

The computation-aware property of the CSS format enables us to then use our new formulation to achieve non-zero memoization, hence sharing intermediate $\mathcal{K}$ vectors across multiple IOU non-zeros and within permutations of a given IOU non-zero. We will explore parallel approaches to computing Eq. (5.12) in the remainder of this section.

5.2 Naive Approach Consider node i_l in the CSS data structure at level l with path $\mathcal{P}_{\mathbf{i}_l} = (i_1, i_2, \dots, i_l)$. Let node $\mathcal{P}_{\mathbf{i}_l}(l)$ store $\mathcal{K}(\mathcal{P}_{\mathbf{i}_l})$. From Eq. (5.10), we note that $\mathcal{K}(\mathcal{P}_{\mathbf{i}_l})$ depends only on $\mathcal{K}(\mathcal{P}'_k), \mathcal{P}'_k = \mathcal{P}_{\mathbf{i}_l} \setminus i_k, 1 \leq k \leq l$. Thus, in order to compute $\mathcal{K}$ at node i_l, we need results from nodes at level $l-1$ that correspond to the last node in paths $\mathcal{P}'$.

A naive implementation of S^{3}TTMc performs a level-wise traversal of CSS to compute the $\mathcal{K}$ for nodes at level l. It memoizes $\mathcal{K}(\mathcal{P})$ at all nodes in level $l-1$, and thus simply retrieves $\mathcal{K}(\mathcal{P}'_k), 1 \leq k \leq l$ to compute $\mathcal{K}(\mathcal{P}_l)$. The memory required for memoization has an upper bound determined by the level l ($1 \leq l < N-2$) with the largest $nnz_l R^l + nnz_{l+1} R^{l+1}$. However, this approach runs into memory bottlenecks because (i) the memory overhead of memoizing becomes large with increasing tensor order, and (ii) while computing $\mathcal{K}(\mathcal{P})$ for nodes at level l, the nodes at level $l-1$ that contribute to the Kronecker product of nodes at level l are not consecutive in memory, and this step becomes quite expensive due to the random accesses in a large $\mathcal{K}(\mathcal{P})$ array. Moreover, the naive approach disregards the ordering of nodes in CSS, due to which it overestimates the memory requirement for memoization. We implement an efficient memoization strategy by only storing the minimum number of intermediate $\mathcal{K}(\mathcal{P})$ vectors needed to perform S^{3}TTMc on the CSS tree while still maintaining computational efficiency, as can be seen from lines 1-4 in the S^{3}TTMc-CSS algorithm (Algorithm 2).

5.3 Optimizations

Memoization overhead We seek to reduce the memory blowup due to non-zero memoization in the naive algorithm. Consider the set of paths $SP(\mathcal{P}_{\mathbf{i}_l}) = \{\mathcal{P}'_k : \mathcal{P}'_k = \mathcal{P}_{\mathbf{i}_l} \setminus i_k, 1 \leq k \leq l\}$ for some path $\mathcal{P}_{\mathbf{i}_l}$ in CSS. From the construction of the CSS format, every path $\mathcal{P}'$ in $SP(\mathcal{P}_{\mathbf{i}_l})$ is either (i) a subpath of $\mathcal{P}_{\mathbf{i}_l}$ or, (ii) node $\mathcal{P}'(l')$ at any level $l' < l$ is to the right of $\mathcal{P}_{\mathbf{i}_l}(l')$, i.e. path $\mathcal{P}'$ is to the right of $\mathcal{P}_{\mathbf{i}_l}$. This property narrows the positions of nodes storing $\mathcal{K}(\mathcal{P}'), \mathcal{P}' \in SP(\mathcal{P}_{\mathbf{i}_l})$ by imposing an ordering to the paths in $SP(\mathcal{P}_{\mathbf{i}_l})$. Thus, we switch to a depth-wise traversal of CSS where, for computing $\mathcal{K}(\mathcal{P}_{\mathbf{i}_{N-1}})$, we store $\mathcal{K}$ vector of only contributing paths in $SP(\mathcal{P}_{\mathbf{i}_l})$ for every level $2 \leq l \leq N-1$ and discard $\mathcal{K}$ vectors of paths that are to the left of $\mathcal{K}(\mathcal{P}_{\mathbf{i}_{N-1}})$. Given a set of IOU non-zeros, the size of memory that needs to be allocated for this memoization strategy is determined by the node p_m at level $N-2$ with the largest number of children. That is, the maximum number of $\mathcal{K}$ vectors that need to be stored before at least one of them is discarded is the sum of the number of $\mathcal{K}$ vectors required to compute $\mathcal{K}(\mathbf{j})$ for every child j of p_m.

Symbolic parent graph Note that since $SP(\mathcal{P}_{\mathbf{i}_l})$ are not disjoint sets, we need to ensure that $\mathcal{K}$ of overlapping elements are not computed multiple times. We resolve this problem by constructing an auxiliary data structure called the *symbolic parent graph* $SG(V,E)$ that retrieves all elements in $SP(\mathcal{P}_{\mathbf{i}_l})$ for every path $\mathbf{i}_l$ in CSS data structure, where

$$V = \bigcup_{l=1}^{N-1} V_l \;\Bigg|\; \begin{aligned} &E = \{(u,v) : u \in V_{j+1}, v \in V_j \cap \\ &\{w : w = \mathcal{P}'(j), \mathcal{P}' \in SP(\mathcal{P}_u)\}, 1 \leq j < N-1\} \end{aligned}$$

Here, V_l is the set of all nodes of the CSS tree at level l, and for every node u in level l, $deg(u) = l$. Moreover, $SG(V,E)$ is a union of nnz_{N-1} directed acyclic graphs, i.e., $SG = \bigcup_{i=1}^{nnz_{N-1}} \mathcal{T}_i^{SG}$, where $\mathcal{T}_i^{SG}$ is the DAG with node i as the first node in its topological sorting. From the construction, we infer that $i \in nz_{N-1}$. The construction of the symbolic parent graphs makes it easy to retrieve the nodes in the CSS tree that contribute to the partial Kronecker product at any given node. The space complexity of this auxiliary graph is $S_{SG} = \beta_{int} \sum_{l=2}^{N-1} l \cdot nnz_l$. The time complexity of $SG(V,E)$ construction is $\mathcal{O}(|V|+|E|) = \mathcal{O}(\sum_{l=2}^{N-1} l \cdot nnz_l)$, which is smaller than the S^{3}TTMc cost by $\mathcal{O}(R^l)$ per CSS level (Eq. (5.14)).

5.4 Parallelism The symbolic parent graph SG is used to perform a parallel depth-wise traversal of CSS to compute the intermediate $\mathcal{K}(\mathcal{P})$ results. The leaf nodes

Algorithm 2: Rank-R S^{3}TTMc-CSS

Result: Matricized S^{3}TTMc, $\mathbf{Y} \in \mathbb{R}^{I \times R^{N-1}}$
Construct symbolic parent graph SG
$p_m = \arg\max_{j \in nz_{N-2}} |child(j)|$
Allocate memoization workspace $WS = \sum_{l=1}^{N-2} R^l \big|\{n : n \in \mathcal{T}_j^{SG}, n \in nz_l \forall j \in child(p_m)\}\big|$
parfor $i_{N-1} = 1, \dots nnz_{N-1}$
  while $u_l \in nodes(\mathcal{T}_{i_{N-1}}^{SG})$ *at level* l **do**
    if $l < 2$ **then**
      $\mathcal{K}(u_l) = \mathbf{U}(u_l)$
    else
      for $\mathcal{P}' \in SP(u_l)$ **do**
        $n = \mathcal{P}_{u_l} \setminus \mathcal{P}'$
        $\mathcal{K}(\mathcal{P}_{u_l}) \leftarrow \mathcal{K}(\mathcal{P}_{u_l}) + \mathbf{U}(n,:) \otimes \mathcal{K}(\mathcal{P}')$
parfor $i_{N-1} = 1, \dots nnz_{N-1}$
  while *ind* $\in$ *nonzero array of node* i_{N-1} **do**
    $\mathcal{P} = (i_1, \dots i_{N-1})$
    AtomicAdd$\big(\mathbf{Y}(ind,:), \mathcal{K}(\mathcal{P}) \cdot \mathcal{X}_{(\mathcal{P},ind)}\big)$

of CSS are distributed over p threads, each of which is responsible for computing $\mathcal{K}(\mathcal{P})$ for all of its leaf nodes, as shown in Algorithm 2. We refer to the set of nodes in level $l-1$ that contribute to $\mathcal{K}(\mathcal{P}_{\mathbf{i}_l})$ of node i_l as the symbolic parents of i_l, i.e., $SP(\mathcal{P}_{\mathbf{i}_l})$. Thread local workspace is allocated based on the strategy described in Sec. 5.3 to reduce memory allocated for memoization.

Thus, we can propagate intermediate Kronecker products to nodes down the levels of the CSS tree using the edges of SG. At the leaf nodes at level $N-1$, the final $\mathcal{K}(\mathcal{P}_{\mathbf{i}_{N-1}})$ vectors are multiplied with the non-zero values at every leaf. Note that while there are no synchronization constraints when computing $\mathcal{K}$ for the nodes of the CSS tree, atomics are required while accumulating results into the matricized output $\mathbf{Y}$.

S^{3}TTMc-CSS can be directly used in Algorithm 1 to gain better performance for Tucker decomposition.

5.5 Cost comparison with baselines We compare the number of floating point operations in S^{3}TTMc-CSS (C^{CSS}) to *SPLATT* (C^{SPLATT}) and *S^{3}TTMc-UCOO* (C^{UCOO}) [2]. We provide a cost model for each of the algorithms, and analyze their performance for rank R-S^{3}TTMc.

Flop count of S^{3}TTMc-CSS We assume $nnz_l \approx \binom{N}{l} unnz$ for $N-1 \geq l \geq \lceil \frac{N}{2} \rceil$. The assumption stems from two observations on the CSS structure - (i) the number of nodes at level l is proportional to the number of distinct subsequences of length l from the set of IOU non-zeros (Sec. 4.1), and (ii) for high-order tensors, frequency of overlap of subsequences between IOU non-zeros reduces deeper down the tree i.e. branching in the tree decreases deeper into the CSS structure. As level $\lceil N/2 \rceil$ has the largest number of nodes, we restrict our assumption to $N-1 \geq l \geq \lceil \frac{N}{2} \rceil$. The cost of computing $\mathcal{K}(\mathcal{P}_{\mathbf{i}_l})$ for any path $\mathcal{P}_{\mathbf{i}_l}$ is the sum of Kronecker products $\mathcal{K}(\mathcal{P}'_k) \otimes \mathbf{U}(i_k,:)$ for each path $\mathcal{P}'_k \in SP(\mathcal{P}_{\mathbf{i}_l}), 1 \leq k \leq l$, as shown in eq. (5.13). We then write the total flop count of S^{3}TTMc computation as the sum of $c(l)$ over all levels of the CSS tree, along with the cost of scaling the $\mathcal{K}$ vectors at level $N-1$ by the non-zero value, as shown in Eq. (5.14).

$$c(l; N, R) = (2l-1) R^l \binom{N}{l} unnz \tag{5.13}$$

$$C^{CSS} = \sum_{l=2}^{N-1} c(l; N, R) + 2 \cdot unnz \cdot N R^{N-1} \tag{5.14}$$

Note that for $l < \lceil \frac{N}{2} \rceil$, we have $nnz_l < nnz_{l+1}$ (Sec. 4), and thus $c(l) < c(l+1)$. For $l \geq \lceil \frac{N}{2} \rceil$, we analyze the extrema of $c(l; N, R)$ to obtain the level l_0 for which C^{CSS} is dominated by $c(l_0; N, R)$.

As $c(l; N, R)$ is log-concave, on inspecting the first derivative of $\log c(l; N, R)$, we have $\log c'(l) > 0$ at $l = \lceil \frac{N}{2} \rceil$ and

$$\frac{d}{dl} \log c(l) \Big|_{l=N-1} = \log R - \left(H_{N-1} - \frac{2N-1}{2N-3} \right) \tag{5.15}$$

where H_n is the n^{th}-harmonic number [43]. We know that H_N closely approximates the natural logarithm function, i.e., $\log N \leq H_N \leq 1 + \log N$. Eq. (5.15) is negative for high-order tensors for suitable values of rank R, as $H_{N-1} - \frac{2N-1}{2N-3}$ is unbounded for large N. This implies that the sign of $(\log c(l))'$ determines if C^{CSS} is dominated by the cost contribution of the last level of CSS, or if there exists a level $N-1 > l_0 \geq \lceil \frac{N}{2} \rceil$ that obtains the maximum $c(l; N, R)$.

Comparison to *S^{3}TTMc-UCOO* *S^{3}TTMc-UCOO* implements Eq. (3.4) without any memoization of intermediate Kronecker products shared between IOU non-zeros. Its cost model is given by $C^{UCOO} = 2R^{N-1} N! \, unnz$.

Irrespective of the level contributing the largest cost to C^{CSS}, it is clear that $\log\left(\frac{C^{CSS}}{C^{UCOO}}\right) < 0$. S^{3}TTMc-CSS performs less work than *S^{3}TTMc-UCOO* for high-order sparse symmetric tensors.

Comparison to *SPLATT* The cost model of *SPLATT* in terms of the number of floating point operations is given by Eq. (5.16).

$$C^{SPLATT} = \sum_{l=2}^{N-1} 2R^{N-l+1} nnz'_l + 2 \cdot unnz \cdot N! \cdot R \tag{5.16}$$

The dominant term in Eq. (5.16) is the computation at level N [34] due to the growth of the factorial term with increasing tensor order.

[2] Since CTF targets sparse symmetric tensor contractions, a more general case compared to TTM, and *S^{3}TTMc-UCOO* outperforms CTF in Sec. 6, we ignore its analysis.

THEOREM 5.1. *S^3TTMc-CSS outperforms SPLATT in terms of number of floating point operations for rank R-S^3TTMc operation on order-N sparse symmetric tensors, if Eq.* (5.15) *is negative.*

Proof. Based on the sign of Eq. (5.15), we compare the flop counts of S^{3}TTMc-CSS and *SPLATT* baseline.

- Eq. (5.15) is negative, implying that there exists a level $N-1 > l_0 \geq \lceil \frac{N}{2} \rceil$ such that $c(l_0; N, R)$ is the dominant term, i.e., $(\log c(l))' = 0$ at $l = l_0$. Then, applying AM-HM inequality, we have

$$\log R \geq \frac{2(2l_0 - N)}{N+1} - \frac{2}{2l_0 - 1} \tag{5.17}$$

  Requiring that $R > 1$ further restricts the feasible values of l_0 to $\frac{N}{2} + 1 \leq l_0 < N - 1$. Using the inequality derived in Eq. (5.17) along with the main condition for this case, i.e., $0 < \log R < H_{N-1} - \frac{2N-1}{2N-3}$, we can write

$$\begin{aligned} \log\left(\frac{C^{CSS}}{C^{SPLATT}}\right) &= \log \frac{(2l_0 - 1)R^{l_0}\binom{N}{l_0} unnz}{RN! \cdot unnz} \\ &< \log \frac{2l_0(N-1)^{l_0 - 1}}{l_0^{l_0}(N - l_0)^{N - l_0}} \end{aligned} \tag{5.18}$$

  Note that Eq. (5.18) is an increasing function in l_0 for $\frac{N}{2} + 1 \leq l_0 < N - 1$ for a given tensor order, and as $N \to \infty$, we see that Eq. (5.18) is negative, i.e., S^{3}TTMc-CSS performs less work than *SPLATT*.

- Eq. (5.15) is non-negative, implying that the dominant term in C^{CSS} is $c(N-1; N, R)$. Then,

$$\begin{aligned} \log \frac{C^{CSS}}{C^{SPLATT}} &= \log \frac{(2N-1)NR^{N-1} \cdot unnz}{RN! \cdot unnz} \\ &> \log\left(\frac{2N-1}{N-1}\right) - \frac{4N-5}{(N-1)(2N-3)} > 0 \end{aligned}$$

  Thus, for $\log R \geq H_{N-1} - \frac{2N-1}{2N-3}$, Algorithm 2 could perform more work than SPLATT.

For S^{3}TTMc, if rank and tensor order satisfy $\log R < H_{N-1} - \frac{2N-1}{2N-3}$, our algorithm is more cost-efficient than *SPLATT*. ∎

Table 2 lists a range of (N, R) pairs using N_{min} and R_{max} values such that for a S^{3}TTMc with tensor order $N \geq N_{min}$ and matrix rank $R \leq R_{max}$, Eq. (5.15) is negative, thus the performance of S^{3}TTMc-CSS is theoretically superior to SPLATT.

R_{max}	2	4	8	16
N_{min}	5	8	14	26

Table 2: S^{3}TTMc-CSS outperforms *SPLATT* when its tensor order $N \geq N_{min}$ and matrix rank $R \leq R_{max}$.

6 Experiments

6.1 Platform and Experimental Configurations

Our experiments are conducted on a shared memory multiprocessor with two 24-core Intel Xeon Gold 6238M processors and 1.5 TBytes memory. The code is written in C++ and parallelized using OpenMP, configured to double-precision floating point arithmetic and 64-bit unsigned integers. It is compiled with GCC 8.3.0 and Netlib LAPACK 3.8.0 for linear algebra routines.

6.2 Datasets

We tested eleven synthetic and two real-world symmetric tensors of varying orders and number of IOU non-zeros, as shown in Table 3. Since real hypergraphs often exhibit strong interactions among groups of nodes, we use this property to generate synthetic hypergraphs with four clusters, with each cluster having a uniform distribution of IOU non-zeros. The *Walmart* and *Amazon* datasets are constructed from real-world hypergraphs, and have been used to evaluate the performance of hypergraph clustering algorithms. *Walmart*, an order-8 symmetric tensor [1], is generated from a hypergraph representing Walmart's user relations where hyperedges are co-purchased products. *Amazon*, an order-14 symmetric tensor [26], is extracted from a user-review hypergraph with product reviews as hyperedges. Only one CSF tree and the fastest leaf-to-root TTMc algorithm outlined in the work [34] are used from SPLATT owing to the symmetry feature. CTF [39, 40] is run with LAPACK 3.8.0 [3] and HPTT 1.0.5 [41] functionalities. The thirteen datasets have been divided into three categories - *small*, *large*, and *huge* - depending on the ability of each implementation to compute S^{3}TTMc successfully within a reasonable time limit. The choice of R for rank-R S^{3}TTMc on each dataset is determined by our cost analysis in Sec. 5.5. Among all datasets in *small* and *large* categories, except for dataset S5, R is set to the largest R_{max}. We chose $R = 3$ for S5 with the purpose to demonstrate a case where *SPLATT* outperforms S^{3}TTMc-CSS and verify Theorem 5.1.

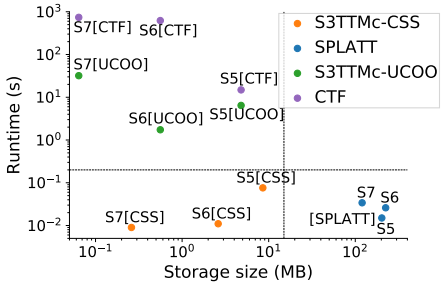
6.3 Overall Performance

We compare the performance of our S^{3}TTMc-CSS with *SPLATT*, *S^3TTMc-UCOO* and *CTF* for all three categories of datasets.

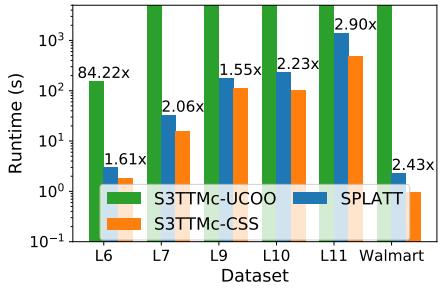
Small datasets Fig. 3a demonstrates the trade-off in the runtime of the S^{3}TTMc operations and the size of the symmetric tensor storage format observed in each of the implementations. The size of the symmetric tensors chosen is small enough to ensure that all baseline implementations run successfully without timing out. We observe three groups in Fig. 3a - (i) *S^3TTMc-UCOO* and CTF implementations, which are the most compact but exhibit high runtimes; (ii) *SPLATT* us-

Table 3: Description of symmetric sparse tensors and implementations that can successfully compute S^{3}TTMc within a reasonable time limit.

Category	Tensor	Order	Dim	*unnz*	*R*	Implementations			
						S^{3}TTMc-UCOO	CTF (*UCOO*)	*SPLATT (FCSF)*	S^{3}TTMc-CSS
Small	S5	5	100	100K	3	✓	✓	✓	✓
	S6	6	100	10K	2	✓	✓	✓	✓
	S7	7	50	1K	3	✓	✓	✓	✓
Large	L6	6	400	1M	2	✗	✗	✓	✓
	L7	7	400	1M	3	✗	✗	✓	✓
	L9	9	400	10K	5	✗	✗	✓	✓
	L10	10	400	1K	5	✗	✗	✓	✓
	L11	11	400	100	6	✗	✗	✓	✓
	Walmart	8	15624	3082	4	✗	✗	✓	✓
Huge	H8	8	400	1M	4	✗	✗	✗	✓
	H12	12	400	10K	3	✗	✗	✗	✓
	H13	13	400	10K	3	✗	✗	✗	✓
	Amazon	14	12981	2131	2	✗	✗	✗	✓



(a)



(b)

Dataset	Size (MB)	Time (s)
H8	1660.00	308.82
H12	578.42	179.07
H13	1150.00	1000.15
Amazon	479.28	15.76

(c)

Figure 3: Comparison of overall performance of S^{3}TTMc-CSS with *SPLATT*, *S^{3}TTMc-UCOO* and CTF for datasets in Table 3. (a) Storage-runtime trade off of all algorithms for *small* tensors. The grouping of runtimes and storage size is similar to Fig. 1. (b) S^{3}TTMc runtime for *large* tensors. *S^{3}TTMc-UCOO* completes only for dataset L6. The speed-up of CSS over each of the baselines is indicated over the corresponding bar. (c) Storage-runtime trade off of S^{3}TTMc-CSS for *huge* tensors. *SPLATT* runs out of memory for this category.

ing *FCSF*, which has a massive memory overhead but is significantly faster than *S^{3}TTMc-UCOO* and CTF using *UCOO*; and (iii) S^{3}TTMc-CSS, which is comparable to *UCOO* in terms of storage requirement while still being faster than *SPLATT*, *S^{3}TTMc-UCOO* and CTF. Fig. 3a aligns perfectly with Fig. 1 and shows that S^{3}TTMc-CSS does an effective trade-off between storage and efficient computation.

S^{3}TTMc-CSS is faster than *SPLATT* for S6 and S7 datasets, as the rank R chosen satisfies Theorem 5.1. However, as we have chosen $R > \exp(H_4 - 1.285)$ (Eq. (5.15)), S^{3}TTMc-CSS performs more work than *SPLATT* and is hence slower.

Large datasets The lack of scalability of both *S^{3}TTMc-UCOO* and CTF with increasing tensor order in Fig. 3a makes them an infeasible baseline for comparison on *large* datasets. While CTF runs out of memory for all datasets, *S^{3}TTMc-UCOO* does not complete in reasonable time for order-7 tensors and higher, as can be seen in Fig. 3b. For the best thread configuration, we achieve up to 2.9× speedup over *SPLATT*, the fastest state-of-the-art baseline (details in Sec. 3). Moreover, our algorithm is always faster than *SPLATT* for *large* symmetric tensors.

Huge datasets Fig. 3c presents the performance of S^{3}TTMc-CSS on *huge* tensors for which *SPLATT* runs out of memory. H8 symmetric tensor has the largest memory requirement at 1.66GB, while S^{3}TTMc on H13 has the longest execution time at 1000 seconds.

6.4 Thread Scalability We analyze thread scalability of S^{3}TTMc-CSS and *SPLATT* for a representative order-10 tensor, L10. Note that we have excluded *S^{3}TTMc-UCOO* and CTF as they do not run to completion for any of the *large* datasets, owing to their lack of scalability with increasing tensor order. *SPLATT* and S^{3}TTMc-CSS show similar scalability due to the similarities in tree layout of the data structure, and its depth wise traversal to compute S^{3}TTMc, as can be seen in Fig. 4. However, S^{3}TTMc-CSS is faster than *SPLATT*, achieving up to 2.46× speedup over *SPLATT* for 8 threads. The scaling becomes poor for higher or-

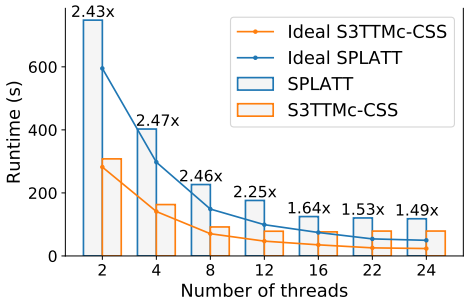


Figure 4: Multithreading scalability of S^{3}TTMc-CSS compared to *SPLATT* for order-10 tensor L10. The speed-up of CSS over *SPLATT* is specified above the blue bars.

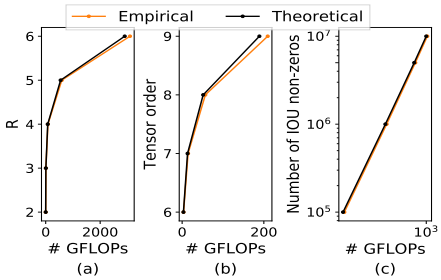


Figure 5: Effect of (a) matrix rank R on S^{3}TTMc-CSS for order-11 tensor L11, (b) tensor order N for rank 2-S^{3}TTMc on tensors having 1M IOU non-zeros, and (c) number of IOU non-zeros in order-7 tensor on S^{3}TTMc-CSS with $R = 3$.

der tensors using more threads, as can be seen in Fig. 4, because of the size of memory allocated per thread increases, resulting in degrading data locality and cache behavior. Another effect of increasing thread memory is the worsening data locality in the Kronecker product computations. Though the memoization strategy described in Algorithm 2 minimizes the number of stored partial Kronecker products, the overhead due to random accesses in the intermediate workspace still remains, and manifests as suboptimal scaling.

6.5 Parameter sweep We analyze the behavior of S^{3}TTMc-CSS as a function of three key parameters affecting the S^{3}TTMc operation - (i) rank R chosen for S^{3}TTMc, (ii) tensor order, and (iii) number of IOU non-zeros in the tensor. We compare the number of floating point operations performed by S^{3}TTMc-CSS with the theoretical number of FLOPs defined in Eq. (5.14). Note that empirical observations closely follow our cost model in Sec. 5.5. Fig. 5a shows the behavior of the S^{3}TTMc operation with varying matrix rank for order-11 tensor, L11. From Eq. (5.14), we see that C^{CSS} grows exponentially with R for a given symmetric tensor, and we correspondingly observe nonlinear scaling in Fig. 5a. This is different from the close to linear scalability of general sparse TTMc [25]. Similarly, a non-linear growth in the number of FLOPs is observed with increasing tensor order (Fig. 5b). We have chosen $R = 2$ - which is R_{max} for order-6 tensors (Table 2) - as it satisfies Theorem 5.1 for all tensor orders considered in the figure. It is difficult to estimate the ideal scaling trend with both R and N from Eq. (5.14) due to the dependence on the level of the CSS tree that dominates the cost of C^{CSS}. For the number of IOU non-zeros *unnz*, Eq. (5.14) indicates linear scaling, the same as observed in Fig. 5c for rank-3 S^{3}TTMc on order-7 tensor. With increasing *unnz*, we also observe non-linear growth in storage size, similar to *FCSF*; though size of *UCOO* increases linearly. There is, however, less significant change in the size of CSS format with increasing mode size I due to the inherent sparsity of the tensor.

6.6 Symmetric Tucker Decomposition We compute symmetric Tucker decomposition on the largest real-world dataset in Table 3, the Amazon dataset. As all the three baselines run out of memory, we use S^{3}TTMc-CSS to compute S^{3}TTMc for the decomposition (Line 2 in Algorithm 1). We are able to run rank-2 symmetric Tucker decomposition for the order-14 Amazon dataset in $\sim$ 1.79hrs, which shows the applicability of this work in the analysis of real world hypergraphs. S^{3}TTMc-CSS provides a practical framework to use symmetric Tucker decomposition in a hierarchical clustering approach to hypergraph analytics.

7 Conclusions

This work focuses on the S^{3}TTMc operation - the specialization of the frequently used tensor times matrix chain operation to symmetric tensors - used in symmetric tensor decomposition. We propose a computation-aware storage format CSS designed for symmetric tensors, with which we can efficiently perform S^{3}TTMc in parallel. We underscore the utility of our algorithm by demonstrating a better trade-off between memory and run-time over SPLATT, S^{3}TTMc-UCOO, and CTF for multiple synthetic and real-world datasets. In the future, we intend to explore the applicability of CSS to other tensor operations like the Matricized Tensor Times Khatri-Rao Product (MTTKRP) and decompositions like tensor train. We plan to improve the access patterns of intermediate results while executing S^{3}TTMc-CSS in parallel, adopt operation memoization, and apply our efficient S^{3}TTMc operation to hypergraph analytics.

Acknowledgments

This research is supported in part by a grant from NVIDIA Corporation.

References

[1] Ilya Amburg, Nate Veldt, and Austin Benson. Clustering in graphs and hypergraphs with categorical edge labels. In *Proceedings of The Web Conference 2020*, pages 706–717, 4 2020.

[2] Animashree Anandkumar, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. Tensor Decompositions for Learning Latent Variable Models. Technical report, 2014.

[3] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK users' guide*. SIAM, 1999.

[4] Joseph Anderson, Mikhail Belkin, Navin Goyal, Luis Rademacher, and James Voss. The More, the Merrier: the Blessing of Dimensionality for Learning Large Gaussian Mixtures. Technical report, 5 2014.

[5] Grey Ballard, Tamara Kolda, and Todd Plantenga. Efficiently computing tensor eigenvalues on a GPU. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1340–1348. IEEE, 5 2011.

[6] Muthu Baskaran, Benoˆit Meister, Richard Lethin, and Jonathan Cai. Optimization of symmetric tensor computations. In *IEEE Conference on High Performance Extreme Computing (HPEC), Waltham, MA, USA*. IEEE, September, Sep 2015.

[7] Jerome Brachat, Pierre Comon, Bernard Mourrain, and Elias Tsigaridas. Symmetric tensor decomposition. *European Signal Processing Conference*, pages 525–529, 1 2009.

[8] Johan Cambré, Lieven De Lathauwer, and Bart De Moor. Best rank-(R, R, R) super-symmetric tensor approximation - A continuous-time approach. In *Proceedings of the IEEE Signal Processing Workshop on Higher-Order Statistics, SPW-HOS 1999*, pages 242–246. Institute of Electrical and Electronics Engineers Inc., 1999.

[9] Pierre Comon, Gene Golub, Lek-Heng Lim, and Bernard Mourrain. Symmetric tensors and symmetric tensor rank. 2 2008.

[10] Ernesto Estrada and Juan A. Rodriguez-Velazquez. Complex Networks as Hypergraphs. *Physica A: Statistical Mechanics and its Applications*, 364:581–594, 5 2005.

[11] Suzanne Renick Gallagher, Micah Dombrower, and Debra S. Goldberg. Using 2-node hypergraph clustering coefficients to analyze disease-gene networks. In *ACM BCB 2014 - 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 647–648, New York, New York, USA, 9 2014. Association for Computing Machinery, Inc.

[12] Debarghya Ghoshdastidar and Ambedkar Dukkipati. Consistency of Spectral Partitioning of Uniform Hypergraphs under Planted Partition Model. Technical report.

[13] Debarghya Ghoshdastidar and Ambedkar Dukkipati. Uniform Hypergraph Partitioning: Provable Tensor Methods and Sampling Techniques. *Journal of Machine Learning Research*, 18(50):1–41, 2017.

[14] Debarghya Ghoshdastidar and Ad@csa Iisc Ernet In. A Provable Generalized Tensor Spectral Method for Uniform Hypergraph Partitioning Ambedkar Dukkipati. Technical report.

[15] Navin Goyal, Santosh Vempala, and Ying Xiao. Fourier PCA and Robust Tensor Decomposition. *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 584–593, 6 2013.

[16] Oguz Kaya and Bora Ucar. High Performance Parallel Algorithms for the Tucker Decomposition of Sparse Tensors. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 103–112. IEEE, 8 2016.

[17] Oguz Kaya and Bora Uçar. Parallel candecomp/parafac decomposition of sparse tensors using dimension trees. *SIAM Journal on Scientific Computing*, 40(1):C99–C130, 2018.

[18] Zheng Tracy Ke, Feng Shi, and Dong Xia. Community Detection for Hypergraph Networks via Regularized Tensor Power Iteration. Technical report, 2020.

[19] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[20] Tamara G. Kolda. Numerical Optimization for Symmetric Tensor Decomposition. 10 2014.

[21] Jiajia Li, Yuchen Ma, and Richard Vuduc. ParTI! : A Parallel Tensor Infrastructure for multicore CPUs and GPUs, 10 2018.

[22] Jiajia Li, Jimeng Sun, and Richard Vuduc. HiCOO: Hierarchical Storage of Sparse Tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 238–252. IEEE, 11 2018.

[23] Jiajia Li, Jimeng Sun, and Richard Vuduc. HiCOO: Hierarchical storage of sparse tensors. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Dallas, TX, USA, November 2018.

[24] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi. A unified optimization approach for sparse tensor operations on GPUs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 47–57, Sept 2017.

[25] Yuchen Ma, Jiajia Li, Xiaolong Wu, Chenggang Yan, Jimeng Sun, and Richard Vuduc. Optimizing sparse tensor times matrix on gpus. *Journal of Parallel and Distributed Computing*, 2018.

[26] Jianmo Ni, Jiacheng Li, and Julian 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)*, pages 188–197, 2019.

[27] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram 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*, SC '19, pages 49:1–49:25, New York, NY, USA, 2019. ACM.

[28] Martin D. Schatz, Tze Meng Low, Robert A. van de Geijn, and Tamara G. Kolda. Exploiting Symmetry in Tensors for High Performance: Multiplication with Symmetric Tensors. *SIAM Journal on Scientific Computing*, 36(5):C453–C479, 1 2014.

[29] Amnon Shashua, Ron Zass, and Tamir Hazan. Multi-way Clustering Using Super-symmetric Non-negative Tensor Factorization. Technical report, 2005.

[30] Decompositionsamantha Sherman and Tamara G Kolda. Estimating Higher-Order Moments Using Symmetric Tensor DecompositionSamanthaDecomposition. Technical report.

[31] Nicholas D. Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E. Papalexakis, and Christos Faloutsos. Tensor Decomposition for Signal Processing and Machine Learning. *IEEE Transactions on Signal Processing*, 65(13):3551–3582, 7 2017.

[32] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications Architectures and Algorithms - IA3 '15*, pages 1–7, New York, New York, USA, 2015. ACM Press.

[33] Shaden Smith and George Karypis. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit. `http://cs.umn.edu/~splatt/`, 2016.

[34] Shaden Smith and George Karypis. Accelerating the tucker decomposition with compressed sparse tensors. In *European Conference on Parallel Processing*. Springer, 2017.

[35] Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, IPDPS, 2015.

[36] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*, 2015.

[37] Edgar Solomonik. *Provably Efficient Algorithms for Numerical Tensor Algebra*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2014.

[38] Edgar Solomonik, James Demmel, and Torsten Hoefler. Communication lower bounds for tensor contraction algorithms. Technical report, ETH Zürich, 2015.

[39] Edgar Solomonik and Torsten Hoefler. Sparse Tensor Algebra as a Parallel Programming Model. Technical report, 2015.

[40] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James 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*, pages 813–824. IEEE, 5 2013.

[41] Paul Springer, Tong Su, and Paolo Bientinesi. Hptt: A high-performance tensor transposition c++ library. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pages 56–62, 2017.

[42] Ledyard R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 9 1966.

[43] Eric W. Weisstein. Harmonic Number.

[44] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with Hypergraphs: Clustering, Classification, and Embedding. Technical report.