Accelerating the Tucker Decomposition with Compressed Sparse Tensors

Shaden Smith(✉) and George Karypis

University of Minnesota, Minneapolis, USA
{shaden,karypis}@cs.umn.edu

Abstract. The Tucker decomposition is a higher-order analogue of the singular value decomposition and is a popular method of performing analysis on multi-way data (*tensors*). Computing the Tucker decomposition of a sparse tensor is demanding in terms of both memory and computational resources. The primary kernel of the factorization is a chain of tensor-matrix multiplications (TTMc). State-of-the-art algorithms accelerate the underlying computations by trading off memory to memoize the intermediate results of TTMc in order to reuse them across iterations. We present an algorithm based on a compressed data structure for sparse tensors and show that many computational redundancies during TTMc can be identified and pruned without the memory overheads of memoization. In addition, our algorithm can further reduce the number of operations by exploiting an additional amount of user-specified memory. We evaluate our algorithm on a collection of real-world and synthetic datasets and demonstrate up to 20.7× speedup while using 28.5× less memory than the state-of-the-art parallel algorithm.

1 Introduction

Tensors, which are the generalization of matrices to higher orders, are a natural way of representing multi-way data (i.e., data which features variables interacting in more than two dimensions). Tensors occupy three or more dimensions (called *modes*) which can represent multi-way interactions between variables. *Tensor factorization* is a technique for enabling structure discovery on multi-way data. The objective of tensor factorization is to model the potentially high-dimensional data in a low rank form that captures the key multi-way interactions found in the data. Tensor factorization is used extensively in areas such as anomaly detection [9], healthcare [29], recommender systems [20], and web search [28]. Common traits among all of these applications are the high dimensionality and extreme level of sparsity of the data.

Tensor factorization takes several forms, with the two most popular being the canonical polyadic decomposition (CPD) and the Tucker decomposition [14]. The CPD has been extensively studied by the HPC community in recent years [12,13,26]. However, the Tucker decomposition, which is computationally more challenging than the CPD, has received relatively less attention. Computing the Tucker decomposition of a sparse tensor is challenging in terms of

F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 653–668, 2017.
DOI: 10.1007/978-3-319-64203-1_47

both time and space. At its core is a tensor-times-matrix chain (TTMc), which multiplies a sparse tensor by dense matrices aligned to all but one of its modes.

Existing strategies for performing TTMc either rely on memoizing intermediate results to save computation [2,11] or operating in a memory-efficient manner at the expense of additional floating-point operations (FLOPs) [15]. The memory overhead of memoization is closely tied to the dimensionality and the sparsity pattern of the tensor, and can result in significant memory overhead. Meanwhile, the memory-efficient strategies require orders of magnitude more computation and are often impractical for large and sparse tensors.

We restructure the underlying computations in order to remove two forms of redundant computations that occur during TTMc. We present an algorithm for performing TTMc with a sparse tensor that is often as computationally efficient as memoized algorithms, while requiring a negligible amount of additional intermediate memory. Our algorithm relies on the recently-developed data structure for tensors called compressed sparse fiber (CSF) [22]. The CSF data structure provides a view of the tensor's sparsity structure that makes these redundancies possible to exploit. Furthermore, we show that an additional, user-specified amount of memory can be used to further reduce computational costs by constructing additional views of the tensor. Our contributions include:

1. A parallel algorithm for TTMc which is memory-efficient while being computationally competitive to the state-of-the-art.
2. An analysis of the TTMc algorithm and demonstration of its potential for asymptotic improvement.
3. A strategy for leveraging multiple compressed tensor representations to further reduce the number of required operations.
4. An experimental evaluation against the state-of-the-art parallel algorithms across a variety of real-world datasets.
5. Integration of our source code into SPLATT [23], an open source library for sparse tensor factorization.

The rest of the paper is organized as follows. Section 2 provides an overview of tensors and tensor factorization. Section 3 reviews related work on TTMc. Section 4 presents and analyzes our algorithm for performing TTMc operations that leverage the sparse tensor representation. Section 5 discusses the benefits of using multiple views of the tensor data and provides a heuristic algorithm for selecting advantageous views. Section 6 presents experimental results. Lastly, Sect. 7 offers concluding remarks.

2 Preliminaries

2.1 Notation

Matrices are denoted using bold letters ($\mathbf{A}$) and tensors using bold calligraphic letters ($\boldsymbol{\mathcal{X}}$). Tensors have N modes with lengths $I_1, \ldots, I_N$, respectively. We denote the number of non-zeros in a tensor as $\text{nnz}(\boldsymbol{\mathcal{X}})$. Entries in matrices and

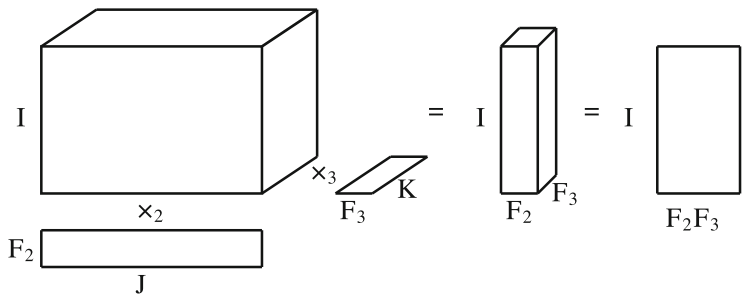


Fig. 1. TTMc with an $I \times J \times K$ tensor. The result is a dense tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I \times F_2 \times F_3}$, which can conceptually be unfolded to $\mathbf{Y}_{(1)} \in \mathbb{R}^{I \times F_2 F_3}$.

tensors are denoted $\mathbf{A}(i,j)$ and $\boldsymbol{\mathcal{X}}(i_1,\ldots,i_N)$, respectively. A colon in the place of an index takes the place of all non-zero entries. For example, $\boldsymbol{\mathcal{X}}(i,j,:,\ldots,:)$ is the set of all non-zeros in $\boldsymbol{\mathcal{X}}$ whose first two indices are (i,j). Similarly, $\mathbf{A}(i,:)$ is the ith row of $\mathbf{A}$. A *fiber* is the generalization of a row or column and is the result of holding all but one index constant (e.g., $\boldsymbol{\mathcal{X}}(i_1,\ldots,i_{N-1},:)$ or $\boldsymbol{\mathcal{X}}(:,i_2,\ldots,i_N)$).

2.2 Tensor and Matrix Operators

Unfolding. Tensors can be "unfolded" along a mode to form a matrix. Unfolding is accomplished by forming columns from the fibers that run along the desired mode. For example, a mode-1 unfolding is denoted $\mathbf{X}_{(1)}$ and has dimension $I_1 \times \prod_{j=2}^{N} I_j$.

Kronecker Product. The Kronecker product (KP) of $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{p \times q}$, denoted $\mathbf{A} \otimes \mathbf{B}$, is an $mp \times nq$ matrix and defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} \mathbf{A}(1,1)\mathbf{B} & \ldots & \mathbf{A}(1,n)\mathbf{B} \\ \vdots & \ddots & \vdots \\ \mathbf{A}(m,1)\mathbf{B} & \ldots & \mathbf{A}(m,n)\mathbf{B} \end{bmatrix}.$$

The KP is a generalization of the vector outer product. Throughout our discussion, we will work in terms of KPs but refer to visualizations of outer products. They are the same operations, but outer products better visualize growth in dimensionality.

Tensor-Matrix Product. The tensor-matrix product, or *n-mode product* [14], multiplies a tensor by a matrix along the nth mode. Suppose $\mathbf{B}$ is an $F \times I_n$ matrix. The tensor-matrix product for the nth mode, denoted $\boldsymbol{\mathcal{X}} \times_n \mathbf{B}$, emits a tensor with dimensions $I_1 \times \ldots \times I_{n-1} \times F \times I_{n+1} \times \ldots \times I_N$. Elementwise,

$$(\boldsymbol{\mathcal{X}} \times_n \mathbf{B})(i_1,\ldots,i_{n-1},f,i_{n+1},\ldots,i_N) = \sum_{i_n=1}^{I_n} \boldsymbol{\mathcal{X}}(i_1,\ldots,i_N)\mathbf{B}(f,i_n).$$

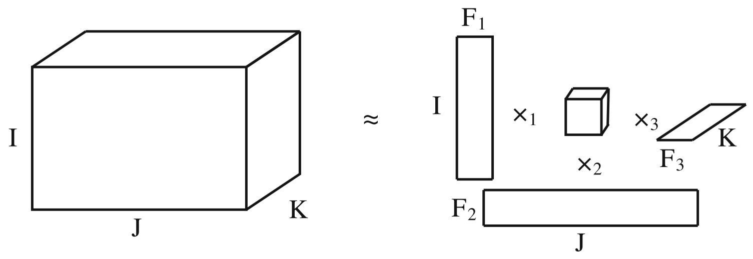


Fig. 2. A rank-$\{F_1, F_2, F_3\}$ Tucker factorization of an $I \times J \times K$ tensor.

Note that the resulting mode-n fibers are generally dense regardless of the sparsity pattern of $\boldsymbol{\mathcal{X}}$.

A common task is to multiply a tensor by a set of matrices. This operation is called the tensor-times-matrix chain (TTMc). When multiplication is performed with all N modes, we write $\boldsymbol{\mathcal{X}} \times \{\mathbf{A}\}$, where $\{\mathbf{A}\}$ is the set of N matrices. More commonly, one wishes to multiply with all modes but one. We write this operation as $\boldsymbol{\mathcal{X}} \times_{-n} \{\mathbf{B}\}$, where n is the mode left unmultiplied:

$$\boldsymbol{\mathcal{X}} \times_{-n} \{\mathbf{B}\} = \boldsymbol{\mathcal{X}} \times_1 \mathbf{B}^{(1)} \times_2 \cdots \times_{n-1} \mathbf{B}^{(n-1)} \times_{n+1} \mathbf{B}^{(n+1)} \times_{n+2} \cdots \times_N \mathbf{B}^{(N)}.$$

This case is the focus of this work, and we will refer to solely it as TTMc for the remaining discussions. TTMc for $n = 1$ is illustrated in Fig. 1. Due to the increasingly dense output of each n-mode product, the size of the intermediate results during TTMc can greatly exceed the size of the inputs or output. This is referred to as the *intermediate blowup problem* [15].

2.3 Tucker Decomposition

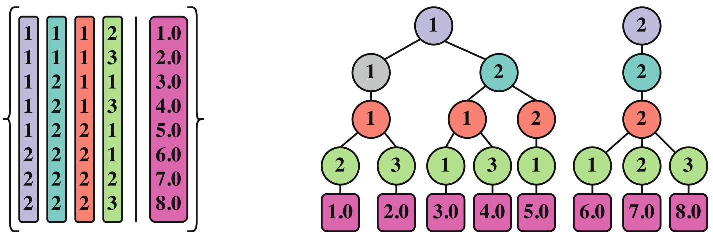
Illustrated in Fig. 2, the objective of the Tucker decomposition is to model a tensor $\boldsymbol{\mathcal{X}}$ with a set of orthonormal matrices $\mathbf{A}^{(1)} \in \mathbb{R}^{I_1 \times F_1}, \ldots, \mathbf{A}^{(N)} \in \mathbb{R}^{I_N \times F_N}$ and a core tensor, $\boldsymbol{\mathcal{G}} \in \mathbb{R}^{F_1 \times \ldots \times F_N}$. The orthonormal matrices are referred to as *factor matrices*. The resulting optimization problem is non-convex:

$$\begin{array}{lll} \underset{\{\mathbf{A}\},\boldsymbol{\mathcal{G}}}{\text{minimize}} & \frac{1}{2}\|\boldsymbol{\mathcal{X}} - \boldsymbol{\mathcal{G}} \times \{\mathbf{A}\}\|_F^2 & \\ \text{subject to} & \mathbf{A}^{(n)T}\mathbf{A}^{(n)} = \mathbf{I} & n = 1, \ldots, N. \end{array}$$

Several optimization algorithms have been developed to compute the Tucker decomposition, including the higher-order SVD (HOSVD) [7] and higher-order orthogonal iterations (HOOI) [8]. HOSVD is popular for decomposing dense tensors and efficient parallel algorithms have been developed [1,5]. However, the computation becomes progressively more dense during HOSVD and it is not often applied to sparse computations. Thus, HOOI is the most popular algorithm for sparse tensors and is the focus of this work. HOOI is an iterative

Algorithm 1. Tucker Decomposition with HOOI

```
1: while $\boldsymbol{\mathcal{G}}$ not converged do
2:     for $n = 1, \ldots, N$ do
3:         $\boldsymbol{\mathcal{Y}} \leftarrow \boldsymbol{\mathcal{X}} \times_{-n} \{\mathbf{A}^T\}$
4:         $\mathbf{A}^{(n)} \leftarrow F_n$ leading left singular vectors of $\mathbf{Y}_{(n)}$
5:     end for
6:     $\boldsymbol{\mathcal{G}} \leftarrow \boldsymbol{\mathcal{Y}} \times_N \mathbf{A}^{(N)T}$
7: end while
```



(a) Coordinate (b) Compressed sparse fiber (CSF)

Fig. 3. Two encodings of a $2 \times 2 \times 2 \times 3$ tensor with 5 non-zeros.

algorithm which cyclically updates each factor matrix until convergence. Algorithm 1 details the steps in computing the factor matrices and core tensor using HOOI. TTMc (Line 3) is the dominant computation during each update.

Most applications involving sparse tensors are not interested in an exact model of a tensor, but instead a low-rank factorization. Therefore, in this work we focus on the case when $\max(F_1, \ldots, F_N) \ll \max(I_1, \ldots, I_N)$.

2.4 Data Structures for Sparse Tensors

The most prevalent data structure for representing sparse tensors is *coordinate format*. Each non-zero is encoded as a tuple of indices and a non-zero value (Fig. 3a). *Dimension trees* are flexible data structures which partition the modes of a tensor in a hierarchical fashion [10]. An important configuration arranges the tensor modes into a binary tree with N leaves [11]. A special case of the dimension tree is the linear arrangement of modes equivalent to coordinate format.

In previous work, we proposed a compressed data structure for sparse tensors called *compressed sparse fiber* (CSF) [22,26]. CSF can be viewed as a generalization of compressed sparse row, a popular storage format for sparse matrices. Shown in Fig. 3b, CSF stores the sparsity pattern as a forest of I_1 trees, each with N levels. Each path from a root to a leaf node encodes a non-zero. The nnz($\boldsymbol{\mathcal{X}}$) leaf nodes store the final index in the non-zero's coordinate and are also accompanied by the non-zero value.

3 Related Work

Li et al. developed parallel algorithms for performing a single TTM kernel for both dense [16] and sparse [17] tensors.

Memory-Efficient Tucker [15] avoids memory blowup by selectively computing columns or elements of $\mathbf{Y}_{(n)}$ at a time. Intermediate memory costs are minimized at the expense of additional FLOPs and passes over the tensor structure.

Baskaran et al. [2] observed that partial computations can be reused across TTMc kernels. Consider updating the first two factors of a four-mode tensor. Each TTMc kernel constructs the partial computation $\boldsymbol{\mathcal{X}} \times_3 \mathbf{A}^{(3)T} \times_4 \mathbf{A}^{(4)T}$, despite its value not changing between kernels. Baskaran et al. introduced memoization to TTMc by partitioning the tensor modes into two halves, and reusing the computations from one half to accelerate the computations in the other half. Kaya and Uçar extended this memoization strategy by using binary dimension trees to accelerate both the Tucker decomposition [11] and CPD [13]. They store intermediate computations in the nodes of the tree and can effectively limit the number of individual n-mode products to $\log(N)$ per TTMc operation.

Kaya and Uçar also showed that one can avoid intermediate blowup by processing individual non-zeros [11]. For example, the following is used for mode-1:

$$\mathbf{Y}_{(1)}(i_1,:) \leftarrow \mathbf{Y}_{(1)}(i_1,:) + \boldsymbol{\mathcal{X}}(i_1,\ldots,i_N)\left[\mathbf{A}^{(2)}(i_2,:) \otimes \cdots \otimes \mathbf{A}^{(N)}(i_N,:)\right]. \quad (1)$$

A row of $\mathbf{Y}_{(1)}$ is the only memory required to process a non-zero. The computational complexity of using (1) to perform one TTMc kernel via streaming through each non-zero is

$$\text{nnz}(\boldsymbol{\mathcal{X}}) \underbrace{\sum_{i=2}^{N}\prod_{j=2}^{i} F_j}_{\text{KP construction}} + \text{nnz}(\boldsymbol{\mathcal{X}}) \underbrace{2\prod_{j=2}^{N} F_j}_{\text{accumulation}} = \mathcal{O}\left(\text{nnz}(\boldsymbol{\mathcal{X}})\prod_{j=2}^{N} F_j\right). \quad (2)$$

4 TTMc with a Compressed Sparse Tensor

We now detail our operation- and memory-efficient parallel algorithm for TTMc. We first perform a reformulation of the underlying computations in order to remove redundancies and then describe a parallel algorithm which uses CSF to exploit these redundancies. We then analyze the computational complexity of our algorithm.

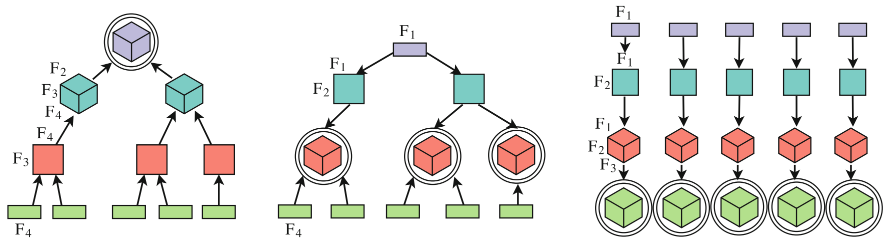
4.1 Formulation

We work from (1) which processes individual non-zeros. There are two forms of arithmetic redundancies that we eliminate during TTMc:

Distributive Kronecker Products. Consider two adjacent non-zeros in a three-mode tensor. Performing a TTMc operation for the first mode results in the following computations:

$$\mathbf{Y}_{(1)}(i,:) \leftarrow \mathbf{Y}_{(1)}(i,:) + \boldsymbol{\mathcal{X}}(i,j,k_1)\left[\mathbf{A}^{(2)}(j,:) \otimes \mathbf{A}^{(3)}(k_1,:)\right], \quad (3a)$$

$$\mathbf{Y}_{(1)}(i,:) \leftarrow \mathbf{Y}_{(1)}(i,:) + \boldsymbol{\mathcal{X}}(i,j,k_2)\left[\mathbf{A}^{(2)}(j,:) \otimes \mathbf{A}^{(3)}(k_2,:)\right]. \quad (3b)$$



(a) CSF (mode-1) (b) CSF (mode-3) (c) Coordinate (mode-4)

Fig. 4. TTMc with CSF and coordinate data structures. The number of FLOPs performed on a node is equal to its volume. Circled nodes produce updates to the output.

The KP is a distributive operation, and so we combine (3a) and (3b) to eliminate a KP and reach a more efficient update:

$$\mathbf{Y}_{(1)}(i,:) \leftarrow \mathbf{Y}_{(1)}(i,:) + \mathbf{A}^{(2)}(j,:) \otimes \left[\boldsymbol{\mathcal{X}}(i,j,k_1)\mathbf{A}^{(3)}(k_1,:) + \boldsymbol{\mathcal{X}}(i,j,k_2)\mathbf{A}^{(3)}(k_2,:)\right].$$

This can be exploited for any set of non-zeros that reside in the same fiber. For each fiber, we accumulate all of the linear combinations of rows of $\mathbf{A}^{(3)}$ into a row vector, followed by a single KP. This eliminates the construction and accumulation of nnz$(\boldsymbol{\mathcal{X}}(i,j,:))-1$ KPs, resulting in a reduction of $2F_2F_3\,(\text{nnz}(\boldsymbol{\mathcal{X}}(i,j,:))-1)$ FLOPs. This strategy generalizes to any number of modes:

$$\mathbf{Y}_{(1)} \leftarrow \sum_{\boldsymbol{\mathcal{X}}(i_1,:,\ldots,:)} \mathbf{A}^{(2)}(i_2,:) \otimes \left(\sum_{\boldsymbol{\mathcal{X}}(i_1,i_2,:,\ldots,:)} \mathbf{A}^{(3)}(i_3,:) \otimes \ldots \left(\sum_{\boldsymbol{\mathcal{X}}(i_1,\ldots,i_{N-1},:)} \boldsymbol{\mathcal{X}}(i_1,\ldots,i_N)\mathbf{A}^{(N)}(i_N,:) \right)\right).$$

Redundant Kronecker Products. Consider the case of performing mode-3 TTMc:

$$\mathbf{Y}_{(3)}(k_1,:) \leftarrow \mathbf{Y}_{(3)}(k_1,:) + \boldsymbol{\mathcal{X}}(i,j,k_1)\left[\mathbf{A}^{(1)}(i,:) \otimes \mathbf{A}^{(2)}(j,:)\right],$$
$$\mathbf{Y}_{(3)}(k_2,:) \leftarrow \mathbf{Y}_{(3)}(k_2,:) + \boldsymbol{\mathcal{X}}(i,j,k_2)\left[\mathbf{A}^{(1)}(i,:) \otimes \mathbf{A}^{(2)}(j,:)\right].$$

Note that $[\mathbf{A}^{(1)}(i,:) \otimes \mathbf{A}^{(2)}(j,:)]$ appears in the processing of both non-zeros. We eliminate operations by reusing the KP for both non-zeros. Reusing the shared KP for an entire fiber saves $F_1F_2(\text{nnz}(\boldsymbol{\mathcal{X}}(i,j,:))-1)$ FLOPs. As before, this process can be generalized to any number of tensor modes.

Operation-Efficient Algorithm. Using the two previous optimizations, we can devise an algorithm which uses the CSF data structure to eliminate redundant operations. A branch in the tree structure at the ith level represents a set of non-zeros which overlap in the previous $i-1$ indices, which is precisely the scenario that the previous optimizations target. Our TTMc algorithm is described in Algorithm 2 and illustrated in Fig. 4. Intuitively, partial computations begin at

Algorithm 2. TTMc with a CSF Tensor (Fig. 4).

```
function TTMC($\boldsymbol{\mathcal{X}}$, $mode$)
    for $i_1 = 1, \ldots, I_N$ in parallel do
        CONSTRUCT($\boldsymbol{\mathcal{X}}(i_1, :, \ldots, :)$, $mode$, $\mathbf{1}$)
    end for
end function
                              ▷ Construct Kronecker products and push them down to level $mode-1$.
function CONSTRUCT($node$, $mode$, $above$)
    $d \leftarrow$ level($node$)                    ▷ The level in the tree (i.e., distance from the root).
    $i_d \leftarrow$ node_id($node$)                 ▷ The partial coordinate of a non-zero (Fig. 3).

    if $d <$ mode then
        $above \leftarrow above \otimes \mathbf{A}^{(d)}(i_d, :)$
        for $c \in$ children($node$) do
            CONSTRUCT($c$, mode, $above$)
        end for

    else if $d =$ mode then
        $below \leftarrow \sum_{c \in \text{children}(node)}$ ACCUMULATE($c$)
        Lock mutex $i_d$.
        $\mathbf{Y}_{(d)}(i_d, :) \leftarrow \mathbf{Y}_{(d)}(i_d, :) + (above \otimes below)$        ▷ Update $\mathbf{Y}_{(d)}$.
        Unlock mutex $i_d$.
    end if
end function
                              ▷ Pull Kronecker products up from the leaf nodes.
function ACCUMULATE($node$)
    $i_d \leftarrow$ node_id($node$)
    if level($node$) $= N$ then
        return $\boldsymbol{\mathcal{X}}(i_1, \ldots, i_d) \cdot \mathbf{A}^{(N)}(i_d, :)$
    else
        return $\mathbf{A}^{(d)}(i_d, :) \otimes \sum_{c \in \text{children}(node)}$ ACCUMULATE($c$)
    end if
end function
```

the root and leaf levels of the tree and grow inward towards the level representing the mode of computation. Algorithm 2 avoids intermediate memory blowup by processing the tree depth-first, which limits the intermediate memory to a single row of $\mathbf{Y}_{(n)}$.

Parallelism. Algorithm 2 is parallelized by distributing the I_1 trees to threads. Each thread performs a depth-first traversal, and thus the thread-local storage overhead is asymptotically limited to a single row of $\mathbf{Y}_{(n)}$. A consequence of this distribution is the potential for write conflicts when updating any modes other than the first. This can be observed in Fig. 3, in which node IDs are only unique within the root-level nodes. The same synchronization challenges are present while computing the CPD, which was the first application of the CSF data structure. We present synchronization using mutexes for simplicity, but note that the algorithm can benefit from other mechanisms such as tiling [22] or transactional memory [25].

4.2 Complexity Analysis

We now analyze the computational complexity of Algorithm 2. Let nodes(i) be the number of nodes present in the ith level of a CSF structure (by convention, the 1st level is the root level). The number of FLOPs required to perform TTMc for the nth mode is $\sum_{i=1}^{N}$ nodes$(i) \times$ cost(i, n), where "cost" is defined as

$$\text{cost}(i,n) = \begin{cases} \prod_{j=1}^{i-1} F_j & \text{if } i < n, \\ 2\prod_{j=i}^{N} F_j & \text{if } i > n, \\ 2\prod_{\substack{j=1 \\ j\neq i}}^{N} F_j & \text{if } i = n. \end{cases} \tag{4}$$

Intuitively, the cost of a node above level-n is the cost of constructing a KP, and the cost at or below level-n is the cost of constructing *and* accumulating a KP.

When computing for the leaf mode of the tensor, Algorithm 2 assembles KPs and pushes them down the tree from root to leaves. The complexity grows with each level of the tree, with the final level having the same asymptotic complexity as (2). At the other extreme, when $n = 1$, the computation moves upwards from leaves to root. Interestingly, the dimensionality of the KPs is non-decreasing, and at the same time the number of nodes in each level is non-increasing. In the worst case, non-zeros have no overlapping indices and the algorithm is equivalent to operating with a tensor stored in coordinate format. However, lower complexities are possible under some assumptions on the CSF structure and the ranks of the factorization. To see, compare the costs of levels i and $i{-}1$:

$$\frac{\text{nodes}(i) \times 2\prod_{j=i}^{N} F_j}{\text{nodes}(i{-}1) \times 2\prod_{j=i-1}^{N} F_j} = \frac{\text{nodes}(i)}{\text{nodes}(i{-}1)F_{i-1}}.$$

Suppose that the cost of the ith mode always exceeds mode $i{-}1$:

$$\text{nodes}(i) > \text{nodes}(i{-}1)F_{i-1}, \qquad i = 2, \ldots, N$$

then the Nth mode dominates the computation, arriving at a reduced complexity of $\mathcal{O}(\text{nodes}(N)F_N) = \mathcal{O}(\text{nnz}(\boldsymbol{\mathcal{X}})F_N)$.

5 Utilizing Additional CSF Representations

Section 4.2 showed that Algorithm 2 has the potential for an asymptotic speedup over the competing memory-efficient approaches. This depends on the costs of the lower levels of the tree dominating those at the top, which is possible if: (i) the branching factor at each level is larger than the corresponding rank; and (ii) the mode on which we are operating is found at or near the top of the tree. Fortunately, CSF places no restriction on the ordering of modes. Indeed, constructing a unique CSF representation for each mode of the tensor was used in other kernels to expose parallelism [26] and to reduce communication costs [24].

Table 1. Summary of datasets.

Dataset	Modes	Non-zeros	Dimensions
`NELL-2` [4]	3	77M	12K, 9K, 29K
`Netflix` [3]	3	100M	480K, 18K, 2K
`Enron` [19]	4	54M	6K, 6K, 244K, 1K
`Alzheimer` [27]	5	6.27M	5, 1K, 156, 1K, 396
`Poisson3D`, `Poisson4D` [6]	3,4	100M	3K, . . . , 3K

K and **M** stand for thousand and million, respectively.

We construct multiple CSF representations in order to minimize the required number of operations. Utilizing multiple CSF representations allows computations to occur near the roots of the tree structures while also favoring mode orderings which result in large branching factors.

There are $N!$ possible orderings of the tensor modes. To evaluate the cost of a representation, we must sort the non-zeros in order to inspect the tree structure and count the number of nodes. Thus, an exhaustive search is impractical for even small values of N. We begin from an existing heuristic: sort the modes by their lengths, with the shortest mode placed at the top level [26]. The intuition behind this heuristic is that ordering shorter modes prior to longer ones discovers indices with high levels of overlap, resulting in a large branching factor.

Suppose there is memory available for up to K representations of the tensor data, denoted $\boldsymbol{\mathcal{X}}_1, \ldots, \boldsymbol{\mathcal{X}}_K$. We select $\boldsymbol{\mathcal{X}}_1$ by sorting the modes as previously discussed. The remaining $K-1$ representations are selected in a greedy fashion: at step k, use (4) to examine the costs associated with TTMc for each mode when provided with $\boldsymbol{\mathcal{X}}_1, \ldots, \boldsymbol{\mathcal{X}}_{k-1}$. The mode with the highest cost is placed at the top level of $\boldsymbol{\mathcal{X}}_k$, and the remaining modes are sorted by increasing length. At the end of this procedure, each mode has the choice of K representations to use for TTMc computation. We assign each mode to the representation with the lowest cost, and use that representation for TTMc. Importantly, if ties are broken in a consistent manner, then it happens in practice that several modes can be assigned to the same $\boldsymbol{\mathcal{X}}_k$, meaning that fewer than K representations need be kept in memory for computation. This is later demonstrated in Sect. 6.2.

6 Experimental Methodology and Results

6.1 Experimental Setup

Experiments are conducted on the Mesabi supercomputer at the Minnesota Supercomputing Institute. Compute nodes have two twelve-core Intel Haswell E5-2680v3 processors and 256 GB of RAM. Our source code is written in C and parallelized with OpenMP. All source code is configured to use double-precision floating point numbers and 32-bit integers. We compile with the Intel compiler version 16.0.3 and Intel MKL for BLAS/LAPACK routines. We bind threads to cores via `KMP_AFFINITY=granularity=fine,compact,1`.

Reported runtimes are the arithmetic mean of twenty iterations. We measure only the time spent on TTMc, as that is the focus of this study and the remaining computational steps do not differ between the implementations. Reported times and speedups are based on performing all of the required computations for TTMc over a full HOOI iteration. Measuring a full HOOI iteration instead of individual kernels allows us to compare memoized and non-memoized algorithms.

We compare against two algorithms implemented in the C++ library HyperTensor [11], the state-of-the-art parallel software for the Tucker decomposition. HyperTensor uses MPI for distributed-memory parallelism and OpenMP for shared-memory parallelism. The efficient distributed-memory algorithm used by HyperTensor combines the communication steps associated with the TTMc and the following truncated SVD, preventing us from measuring the runtime corresponding to only TTMc. Thus, we run HyperTensor with one MPI rank and twenty-four OpenMP threads. We denote the two algorithms as HT-FLAT, which is a direct implementation of (1), and HT-BTREE, which uses memoization via binary dimension trees.

Datasets. Table 1 provides an overview of the datasets used in our evaluation. `NELL-2` is from the Never Ending Language Learning project [4] and its modes represent *entities*, *relations*, and *entities*. `Netflix` [3] is constructed from movie ratings and has modes representing *users*, *movies*, and *dates*. `Enron` [19] is parsed from an email corpus spanning three years. Its non-zero values are word frequency and its modes represent *senders*, *receivers*, *words*, and *dates*. `Alzheimer` is constructed from public gene expression data related to Alzheimer's disease, provided by MSigDB [27]. Its values are binary and its five modes represent cell type, drug, binned dosage, gene, and binned amplitudes. `Poisson` is a set of synthetically-generated tensors whose values follow a Poisson distribution. We generated tensors following the method of Chi and Kolda [6] with three and four modes of length 3000 and 100-million non-zeros. All tensors except `Netflix` and `Alzheimer` are freely available as part of the FROSTT collection [21].

6.2 Results

Operation Efficiency. Figure 5 shows the number of FLOPs required to perform TTMc. HT-FLAT (coordinate format) is used as a baseline because a CSF tensor will match its complexity if it achieves no compression.

A single CSF representation (CSF-1) reduces computational costs by $59\% - 83\%$ compared to the baseline. Interestingly, CSF-1 is nearly identical in cost to the memoized HT-BTREE algorithm on the three-mode datasets. This is due to the limited amount of memoization possible for a three-mode tensor: one TTMc is computed at full cost and is used to optimize the remaining two operations. This matches the limitation of CSF-1, in which the leaf-level mode must still be computed at full cost. Optimizing for the leaf mode by using CSF-2 is sufficient to achieve the best-possible FLOP performance on all three-mode tensors.

Both HT-BTREE and the CSF variants improve over HT-FLAT as the number of modes increase, because additional tensor modes bring additional TTMc

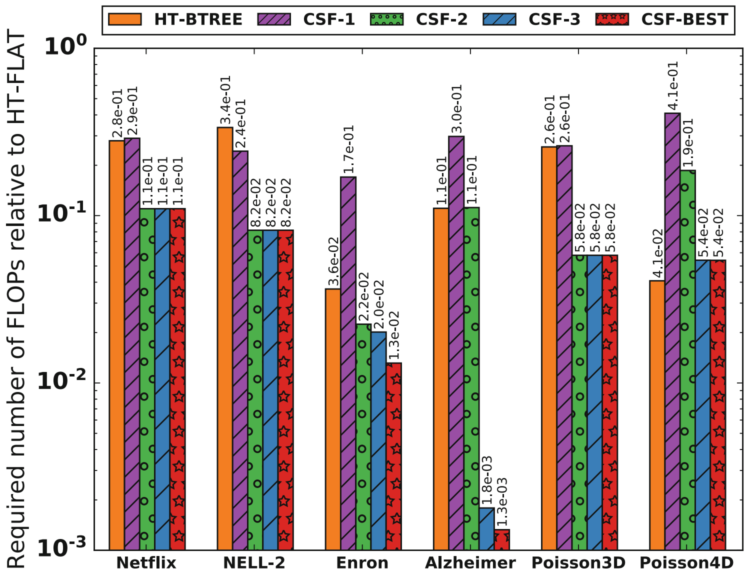


Fig. 5. The number of required FLOPs for rank-20 TTMc on all modes, relative to HT-FLAT (i.e., coordinate form). **CSF-**X is the solution found using X CSF representations. No dataset utilized more than three CSF representations. **CSF-BEST** is the optimal configuration using multiple CSF representations, found by exhaustive search.

operations which can be optimized. The benefits of CSF are most apparent on the five-mode `Alzheimer` tensor, in which the greedily-selected CSF-A requires 555× fewer FLOPs than HT-FLAT and 61× fewer FLOPs than HT-BTREE.

Observe that HT-BTREE is more operation-efficient than CSF-based methods on the synthetic `Poisson4D` tensor. The number of $\boldsymbol{\mathcal{X}}(i_1, i_2, :, \ldots, :)$ subtensors is 88% of the total number of non-zeros, meaning that the redundancies that CSF exploits do not exist in the lower levels of the tree.

Parallel Scalability. Figure 6 shows speedup as we scale from 1 to 24 cores. We include results for CSF-A which dedicates a CSF representation for each mode of the tensor, despite fewer representations being sufficient in terms of FLOP efficiency. CSF-A allows us to measure performance without fine-grained synchronization overheads because there are no race conditions to consider when the output mode is located at the root level of the tree.

Synchronization overheads prevent CSF-1 from scaling beyond one CPU socket, whereas additional CSF representations achieve near-linear scaling. The cost of synchronization dominates when computing for the bottom levels of the CSF structure: there are more nodes present in the tree (i.e., more synchronizations) and also the amount of work performed during synchronizations exponentially increases.

All methods exhibit poor scalability on the `Alzheimer` tensor. This is attributed to its unusually short dimensions; the presented methods parallelize

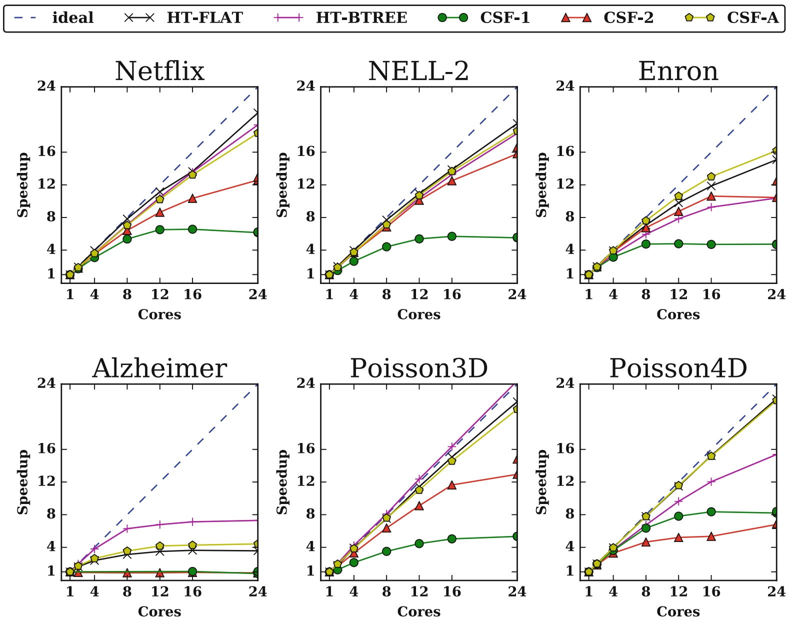


Fig. 6. Parallel speedup for rank-20 TTMc. **CSF-A** denotes dedicating one CSF representation for each mode of the tensor.

over the outer dimensions of the tensor and thus have idle threads when the outer dimension is small. This limitation has also been observed in other tensor kernels [18], and has been remedied via alternative parallel decompositions [2,25]. Exploring these alternative decompositions is left to future work.

Runtime and Memory Trade-Offs. Figure 7 shows the memory costs and average runtime for TTMc. We measure memory consumption via instrumented source code which tracks the storage used for the tensor structure, thread-local storage, and memoization. We omit the storage dedicated to the factor matrices and output because they are the same between methods.

Despite CSF-A not providing additional computational savings, we can see that it always achieves the best runtime across all datasets and algorithms. This is expected due to its lack of synchronization overheads and structured writes to memory. CSF-A ranges from $1.5\times - 20.7\times$ faster than HT-BTREE, and also uses less memory for four of the six datasets. We note that while `Poisson4D` is the only tensor for which memoization achieves a better operation reduction than the CSF variants, but CSF-A is $1.5\times$ faster in runtime.

We can see the benefit of supporting a flexible number of CSF representations. CSF-1 is always the most space-efficient, while CSF-A is always the fastest algorithm. CSF-2 provides a reasonable trade-off when time and space are both limited by dedicating a special CSF representation to the most expensive mode which will also exhibit the highest synchronization costs.

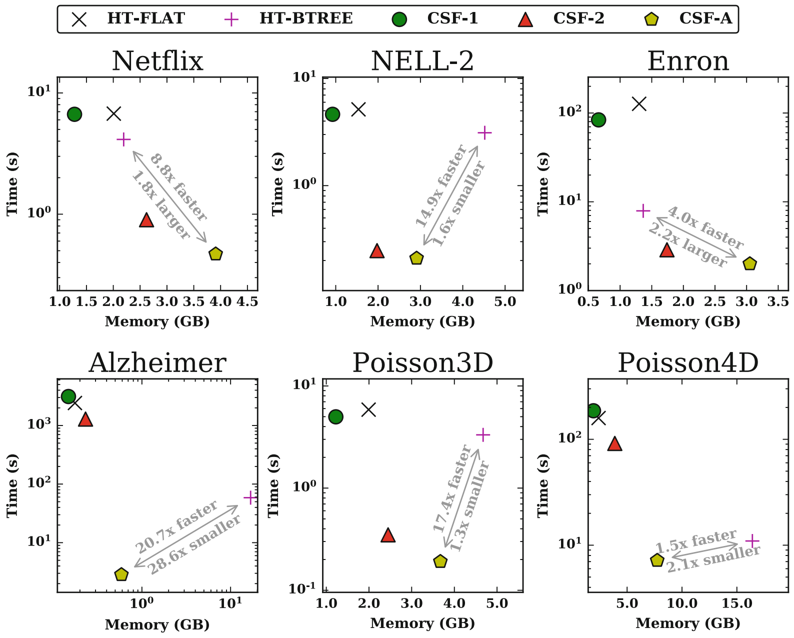


Fig. 7. Time and space trade-offs for rank-20 TTMc on 24 cores. **Time** is the mean number of seconds spent on TTMc during a full iteration of HOOI. **Memory** is the storage required for the tensor memoization, and structures for parallelism.

7 Conclusions and Future Work

A sparse tensor-times-matrix chain (TTMc) is the key computational kernel when computing the Tucker decomposition, which is an important technique for analyzing sparse tensors. We presented a formulation, complexity analysis, and performance evaluation for performing sparse TTMc with a compressed data structure (CSF). We showed that our formulation is both memory-efficient and can be asymptotically faster than competing methods. Our performance evaluation demonstrated up to 20× speedup over the state-of-the-art while at the same time using 28× less memory on a real-world dataset. This effectively reduces the time-to-solution from several hours to a few minutes on a workstation.

Furthermore, we presented a method of tuning the trade-off between the time and memory footprint of the computation. Users can have either the fastest execution, the smallest memory footprint, or in-between the two extremes.

There are several topics of future work. One major advantage of multiple CSF representations is the enhanced scalability via eliminated mutexes. Other CSF algorithms have had success with techniques such as tiling [22, 25] or transactional memory [25], and we will investigate their benefits on TTMc. Alternative parallel decompositions (such as tiling) are also expected to improve parallel scalability on tensors such as `Alzheimer`, which presented difficulties for all methods. Lastly, our cost model could be improved by considering synchronization costs.

Acknowledgments. The authors would like to thank Oguz Kaya for sharing the HyperTensor source code, Muthu Baskaran for providing the `Alzheimer` tensor, Jee W. Choi for providing the synthetic tensor generator, and anonymous reviewers for their valuable feedback. This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), a University of Minnesota Doctoral Dissertation Fellowship, Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

References

1. Austin, W., Ballard, G., Kolda, T.G.: Parallel tensor compression for large-scale scientific data. In: International Parallel and Distributed Processing Symposium (IPDPS'17), pp. 912–922. IEEE (2016)
2. Baskaran, M., Meister, B., Vasilache, N., Lethin, R.: Efficient and scalable computations with sparse tensors. In: 2012 IEEE Conference on High Performance Extreme Computing (HPEC), pp. 1–6. IEEE (2012)
3. Bennett, J., Lanning, S.: The netflix prize. In: Proceedings of KDD cup and workshop, vol. 2007, p. 35 (2007)
4. Carlson, A., Betteridge, J., Kisiel, B., Settles, B., Hruschka, E.R., Mitchell, T.M.: Toward an architecture for never-ending language learning. In: In AAAI (2010)
5. Chakaravarthy, V.T., Choi, J.W., Joseph, D.J., Liu, X., Murali, P., Sabharwal, Y., Sreedhar, D.: On optimizing distributed tucker decomposition for dense tensors. In: 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017) (2017)
6. Chi, E.C., Kolda, T.G.: On tensors, sparsity, and nonnegative factorizations. SIAM J. Matrix Anal. Appl. **33**(4), 1272–1299 (2012)
7. De Lathauwer, L., De Moor, B., Vandewalle, J.: A multilinear singular value decomposition. SIAM J. Matrix Anal. Appl. **21**(4), 1253–1278 (2000a)
8. De Lathauwer, L., De Moor, B., Vandewalle, J.: On the best rank-1 and rank-(r 1, r 2,., rn) approximation of higher-order tensors. SIAM J. Matrix Anal. Appl. **21**(4), 1324–1342 (2000b)
9. Fanaee-T, H., Gama, J.: Tensor-based anomaly detection: an interdisciplinary survey. Knowl.-Based Syst. **98**, 130–147 (2016)
10. Grasedyck, L.: Hierarchical singular value decomposition of tensors. SIAM J. Matrix Anal. Appl. **31**(4), 2029–2054 (2010)
11. Kaya, O., Uçar, B.: High-performance parallel algorithms for the tucker decomposition of higher order sparse tensors. Technical report (2015a)
12. Kaya, O., Uçar, B.: Scalable sparse tensor decompositions in distributed memory systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 77. ACM (2015b)
13. Kaya, O., Uçar, B.: Parallel CP decomposition of sparse tensors using dimension trees. Research report RR-8976, Inria - Research Centre Grenoble - Rhône-Alpes, November 2016

14. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM Rev. **51**(3), 455–500 (2009)
15. Kolda, T.G., Sun, J.: Scalable tensor decompositions for multi-aspect data mining. In: 2008 Eighth IEEE International Conference on Data Mining, ICDM 2008, pp. 363–372. IEEE (2008)
16. Li, J., Battaglino, C., Perros, I., Sun, J., Vuduc, R.: An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 76. ACM (2015)
17. Li, J., Ma, Y., Yan, C., Vuduc, R.: Optimizing sparse tensor times matrix on multi-core and many-core architectures. In: Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms, pp. 26–33. IEEE Press (2016)
18. Rolinger, T.B., Simon, T.A., Krieger, C.D.: Performance evaluation of parallel sparse tensor decomposition implementations. In: Proceedings of the 6th Workshop on Irregular Applications: Architectures and Algorithms. IEEE (2016)
19. Shetty, J., Adibi, J.: The enron email dataset database schema and brief statistical report. Information Sciences Institute Technical report 4, University of Southern California (2004)
20. Shi, Y., Karatzoglou, A., Baltrunas, L., Larson, M., Hanjalic, A., Oliver, N.: TFMAP: optimizing MAP for top-n context-aware recommendation. In: Proceedings of the 35th international ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 155–164. ACM (2012)
21. Smith, S., Choi, J.W., Li, J., Vuduc, R., Park, J., Liu, X., Karypis, G.: FROSTT: the formidable repository of open sparse tensors and tools (2017a). http://frostt.io/
22. Smith, S., Karypis, G.: Tensor-matrix products with a compressed sparse tensor. In: 5th Workshop on Irregular Applications: Architectures and Algorithms (2015)
23. Smith, S., Karypis, G.: SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit (2016). http://cs.umn.edu/splatt/
24. Smith, S., Park, J., Karypis, G.: An exploration of optimization algorithms for high performance tensor completion. In: Proceedings of the 2016 ACM/IEEE conference on Supercomputing (2016)
25. Smith, S., Park, J., Karypis, G.: Sparse tensor factorization on many-core processors with high-bandwidth memory. In: 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS 2017) (2017b)
26. Smith, S., Ravindran, N., Sidiropoulos, N.D., Karypis, G.: SPLATT: efficient and parallel sparse tensor-matrix multiplication. In: International Parallel and Distributed Processing Symposium (IPDPS 2015) (2015)
27. Subramanian, A., Tamayo, P., Mootha, V.K., Mukherjee, S., Ebert, B.L., Gillette, M.A., Paulovich, A., Pomeroy, S.L., Golub, T.R., Lander, E.S., et al.: Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. Proc. Nat. Acad. Sci. **102**(43), 15545–15550 (2005)
28. Sun, J.T., Zeng, H.J., Liu, H., Lu, Y., Chen, Z.: Cubesvd: a novel approach to personalized web search. In: Proceedings of the 14th International Conference on World Wide Web, pp. 382–390. ACM (2005)
29. Wang, Y., Chen, R., Ghosh, J., Denny, J.C., Kho, A., Chen, Y., Malin, B.A., Sun, J.: Rubik: knowledge guided tensor factorization and completion for health data analytics. In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1265–1274. ACM (2015)