Tensor-Matrix Products with a Compressed Sparse Tensor

Shaden Smith
shaden@cs.umn.edu

George Karypis
karypis@cs.umn.edu

Department of Computer Science and Engineering
University of Minnesota, USA

ABSTRACT

The Canonical Polyadic Decomposition (CPD) of tensors is a powerful tool for analyzing multi-way data and is used extensively to analyze very large and extremely sparse datasets. The bottleneck of computing the CPD is multiplying a sparse tensor by several dense matrices. Algorithms for tensor-matrix products fall into two classes. The first class saves floating point operations by storing a compressed tensor for each dimension of the data. These methods are fast but suffer high memory costs. The second class uses a single uncompressed tensor at the cost of additional floating point operations. In this work, we bridge the gap between the two approaches and introduce the *compressed sparse fiber* (CSF) a data structure for sparse tensors along with a novel parallel algorithm for tensor-matrix multiplication. CSF offers similar operation reductions as existing compressed methods while using only a single tensor structure. We validate our contributions with experiments comparing against state-of-the-art methods on a diverse set of datasets. Our work uses 58% less memory than the state-of-the-art while achieving 81% of the parallel performance on 16 threads.

1. INTRODUCTION

Extending a matrix to span more than two dimensions results in a *tensor*. Tensors are data structures that represent multi-way data that is found in many modern applications. Recently, tensors have become popular in communities such as data mining, recommender systems, and health informatics [8, 10, 17]. Tensors in these domains can have dimensions (called *modes*) with tens of millions of features and are very sparse. One example is a ratings matrix extended to include contextual information, resulting in *user-item-context* tuples associated with item ratings. Those tuples naturally form a three-mode tensor.

Tensor factorization is a powerful tool for analyzing multi-way data. The Canonical Polyadic Decomposition (CPD), also known as PARAFAC, is a factorization that has long been used in fields such as psychometrics [5] and chemistry [12]. The CPD is a generalization of the singular value decomposition, outputting a matrix factor for each mode of the tensor. Tensor factorization can be used to find a low-rank representation of sparse data, which provides insights not usually obvious in the original dimensionality.

A common algorithm for finding the CPD is the method of alternating least squares (ALS), which approximates the non-convex optimization problem with a series of convex least squares solutions and iterates until some convergence criteria is met. At the core of each least squares problem is a tensor-matrix product, in which the sparse tensor is multiplied by the dense matrix factors. Every ALS iteration executes a tensor-matrix product for each mode of the tensor.

The mode-centric nature of the computations is a major challenge when designing high-performance tensor algorithms. Optimized algorithms such as SPLATT [19] and DFacTo [6] rely on a different compressed representation of the tensor for each mode in order to reduce floating point operations (FLOPs) and to extract parallelism. The high memory footprint of the optimized methods imposes size constraints on the input tensor, especially in the number of modes represented. The alternative approach is to use a single representation of the tensor in its original tuple form. This approach trades a smaller memory footprint for missed opportunities for parallelism and fewer floating-point operations.

In this work, we address the memory/computation trade-off by extending SPLATT to operate on a single compressed tensor. Our adapted data structure, called *compressed sparse fiber* (CSF), works with an arbitrary number of modes and allows us to perform tensor-matrix products with intermediate memory constant in the number of modes and the rank of the factorization. We show that tiling over the sparsity structure removes the need for mutexes and sophisticated synchronization on shared memory systems. In summary, our contributions are:

1. Serial and shared-memory parallel algorithms for tensor-matrix multiplication that operate on a single tensor representation stored in CSF.
2. A method of tiling over a sparse tensor that avoids locking during parallel computation.
3. An extensive set of experiments evaluating our method against state-of-the-art approaches. This evaluation

IA^3 2015, November 15-20, 2015, Austin, TX, USA.

ACM 978-1-4503-4001-4/15/11 ...$15.00.
DOI: http://dx.doi.org/10.1145/2833179.2833183

shows that on average, our parallel algorithm requires 58% less memory than the best state-of-the-art approach while achieving 81% of the performance on 16 threads.

The rest of this paper is organized as follows. Section 2 introduces notation and includes a brief background on tensor factorization. Section 3 discusses state-of-the-art approaches to tensor-matrix multiplication. Section 4 introduces the CSF data structure and associated serial algorithms, and Section 4.3 describes their parallelization. Our experimental evaluation is discussed in Section 5 and results are discussed in Section 6. Lastly, we make some concluding remarks and discuss future work in Section 7.

2. BACKGROUND AND NOTATION

We denote matrices as $\mathbf{A}$ and tensors as $\boldsymbol{\mathcal{X}}$. The nonzero element with coordinate (i, j, k) is $\boldsymbol{\mathcal{X}}(i, j, k)$. A tensor $\boldsymbol{\mathcal{X}}$ has m modes and $\text{nnz}(\boldsymbol{\mathcal{X}})$ nonzero values. A colon in the place of an index represents all members of that mode. For example, $\mathbf{A}(:, f)$ is column f of matrix $\mathbf{A}$. *Fibers* are the generalization of matrix rows and columns and are the result of holding all but one index constant. One example is $\boldsymbol{\mathcal{X}}(i, j, :, l)$. *Slices* are the result of holding all but two indices constant, creating a matrix.

A tensor can be unfolded, or *matricized*, into a matrix along any of its modes. In the mode-n matricization, the mode-n fibers form the columns of the resulting matrix. The mode-n unfolding of $\boldsymbol{\mathcal{X}}$ is denoted as $\mathbf{X}_{(n)}$. If $\boldsymbol{\mathcal{X}}$ has dimensions $I \times J \times K \times L$, then $\mathbf{X}_{(2)}$ is of dimension $J \times IKL$.

There are two essential matrix operations used in CPD. The *Hadamard product*, denoted by $\mathbf{A} * \mathbf{B}$, is the element-wise multiplication of $\mathbf{A}$ and $\mathbf{B}$. The *Khatri-Rao product*, denoted by $\mathbf{A} \odot \mathbf{B}$, is the column-wise Kronecker product. If $\mathbf{A}$ is $I \times J$ and $\mathbf{B}$ is $M \times J$, then $\mathbf{A} \odot \mathbf{B}$ is $IM \times J$.

2.1 Canonical Polyadic Decomposition

Let the tensor $\boldsymbol{\mathcal{X}}$ have m modes and dimensions $I_1 \times \ldots \times I_m$. The CPD of rank-$F$ seeks to decompose $\boldsymbol{\mathcal{X}}$ into matrix factors $\mathbf{A}_1 \in \mathbb{R}^{I_1 \times F}, \ldots, \mathbf{A}_m \in \mathbb{R}^{I_m \times F}$. Outer products of the F matrix columns form rank-1 tensors approximating $\boldsymbol{\mathcal{X}}$:

$$\boldsymbol{\mathcal{X}} \approx \sum_{f=1}^{F} \mathbf{A}_1(:, f) \circ \cdots \circ \mathbf{A}_m(:, f).$$

We are almost always interested in $F \ll \max\{I_1, \ldots, I_m\}$ for sparse tensors and consider F to be a small constant on the order of 10 or 100.

The CPD is computed by solving the following non-convex optimization problem:

$$\underset{\mathbf{A}_1, \ldots, \mathbf{A}_m}{\text{minimize}} \qquad ||\mathbf{X}_{(1)} - \mathbf{A}_1 \left(\mathbf{A}_m \odot \cdots \odot \mathbf{A}_2\right)^\intercal ||_F.$$

A popular method is to use ALS, which is an iterative algorithm that solves the problem with a series of convex solutions. It is based on the observation that if we hold all but one variable constant, the problem has a least squares solution. During ALS we hold all factors constant and cyclicly update $\mathbf{A}_1, \ldots, \mathbf{A}_m$. This process is repeated until we converge on a local minimum.

Algorithm 1 Computing the CPD with ALS

```
1: while not converged do
2:     for i ∈ {1, 2, ..., m} do        ▷ Iterate over all modes
3:         M ← 1^{F×F}
4:         for j ∈ {1, 2, ..., m} \ {i} do
5:             M ← M * (A_j^T A_j)
6:         end for
7:         Â_i ← X_(i) (A_m ⊙ ··· ⊙ A_{i+1} ⊙ A_{i−1} ⊙ ··· ⊙ A_1)
8:         A_i ← Â_i (M^{−1})
9:         Normalize columns of A_i
10:    end for
11: end while
```

Algorithm 1 shows the steps performed in ALS. The bulk of the work is performed in line 7, and the operation associated with it is often called the *matricized tensor times Khatri-Rao product* (MTTKRP). Explicitly forming the Khatri-Rao product results in a dense matrix of dimension $\prod_{j \neq i} I_j \times F$, requiring orders of magnitude more memory than the original sparse tensor. Instead, the block structure of the Khatri-Rao product is exploited to perform the multiplication *in place*. The fastest MTTKRP algorithms can execute MTTKRP in $O(F \cdot \text{nnz}(\boldsymbol{\mathcal{X}}))$ FLOPs, with the leading constant dependent on the sparsity pattern of the tensor [6, 16, 19].

We refer the reader to the survey by Kolda and Bader [11] for a more thorough discussion of tensor factorization.

3. RELATED WORK

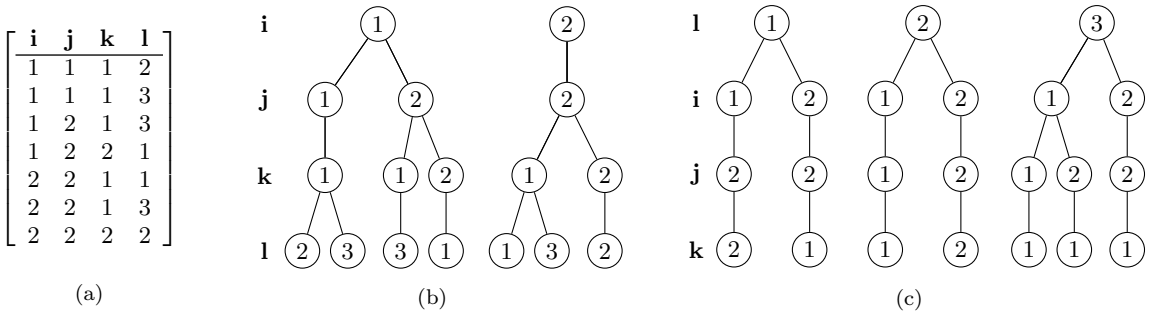
MTTKRP algorithms fall into two classes. The first class stores tensors as an uncompressed list of nonzeros and their coordinates. During MTTKRP, each nonzero in $\boldsymbol{\mathcal{X}}$ contributes to $\hat{\mathbf{A}}$ via

$$\hat{\mathbf{A}}_1(i, f) \leftarrow \boldsymbol{\mathcal{X}}(i, j, \ldots, l) \left[\mathbf{A}_2(j, :) * \cdots * \mathbf{A}_m(l, f)\right]. \qquad (1)$$

Approaches following Equation (1) use $mF \cdot \text{nnz}(\boldsymbol{\mathcal{X}})$ FLOPs for MTTKRP. Tensor Toolbox [1, 2] is a widely used Matlab package for dense and sparse tensor operations. Sparse tensors are stored in coordinate format and MTTKRP requires $O(\text{nnz}(\boldsymbol{\mathcal{X}}))$ intermediate storage. HyperTensor [9] is a recent work for computing the CPD on distributed-memory systems. It uses hypergraph partitioning to distribute nonzeros while minimizing communication. The tensor is uncompressed and its MTTKRP algorithm is a direct implementation of Equation (1).

The second class of MTTKRP algorithms uses compressed structures for each mode of $\boldsymbol{\mathcal{X}}$. The compressed structure reduces the number of operations required during MTTKRP by factoring out redundant multiplications. DFacTo [6] is a CPD algorithm for three-mode tensors on distributed-memory systems. Each of the three compressed tensors is stored as two compressed sparse row (CSR) matrices. Columns of $\hat{\mathbf{A}}$ are computed one at a time via two sparse matrix-vector multiplications. DFacTo executes $2F(P + \text{nnz}(\boldsymbol{\mathcal{X}}))$ FLOPs, where P is the number of non-empty $\boldsymbol{\mathcal{X}}(i, :, k)$ fibers.

Ravindran et al. introduced an alternative approach to MTTKRP for 3D tensors [16]. The authors reformulated the



(a) (b) (c)

Figure 1: (a) $\boldsymbol{\mathcal{X}}$, a 2×2×2×3 sparse tensor stored in coordinate form. (b) The mode-1 tree of $\boldsymbol{\mathcal{X}}$. (c) The mode-4 tree of $\boldsymbol{\mathcal{X}}$.

tensor-matrix products into a series of matrix-matrix products with the frontal $\boldsymbol{\mathcal{X}}(:,:,k)$ slices, resulting in a single access pattern of the tensor at the cost of $O(\text{nnz}(\boldsymbol{\mathcal{X}}))$ intermediate memory. This work was mostly theoretical and left implementation, parallelization, and evaluation to future work.

3.1 SPLATT

Our prior work, called SPLATT [18,19], is a parallel toolkit for sparse tensors on shared- and distributed-memory systems. SPLATT is implemented for 3D tensors and stores each representation in a data structure extended from CSR. Consider the case of storing the mode-1 representation of a tensor. Tensor nonzeros are sorted into $\boldsymbol{\mathcal{X}}(i,j,:)$ fibers. Fibers are stored as sparse vectors via two arrays: an integer array storing the mode-3 indices and an array of the floating-point values. Non-empty fibers are given integer keys storing the mode-2 indices and then grouped by their mode-1 indices into $\boldsymbol{\mathcal{X}}(i,:,:)$ slices. Tensor slices can be thought of a matrix in CSR whose rows are the non-empty fibers.

DFacTo and SPLATT ultimately execute the same number of FLOPs, but feature different memory access patterns. Contrary to the column-centric computation of DFacTo, SPLATT computes whole rows of $\hat{\mathbf{A}}$ at a time by processing all of the fibers in a single slice together:

$$\hat{\mathbf{A}}_1(i,:) \leftarrow \sum_{j=1}^{I_2} \left(\mathbf{A}_2(j,:) * \sum_{k=1}^{I_3} \boldsymbol{\mathcal{X}}(i,j,k)\mathbf{A}_3(k,:) \right).$$

Fibers in distinct slices can be processed in parallel, resulting in *coarse-grained* parallelism. SPLATT distributes whole slices to threads without worries of race conditions. Since updates are done at the granularity of matrix rows, only F words of intermediate memory per thread are needed.

4. MEMORY-EFFICIENT MTTKRP

We will now detail the data structure and algorithms developed in this work to achieve memory-efficient, parallel computations.

4.1 Compressed Sparse Fiber

Our objective is to design a data structure for sparse tensors which reduces memory consumption, can be used for all modes of MTTKRP, and can be efficiently parallelized on multi-core architectures.

Suppose we have a sparse tensor with dimensions $I_1 \leq \cdots \leq I_m$. We form a tree with m levels from each of the $\boldsymbol{\mathcal{X}}(i,:,\ldots,:)$ subtensors. Nodes in level l contain indices found in the lth mode. Paths from root to leaf encode a nonzero coordinate and their values are also stored in the leaves. Figure 1 is an example of a four-mode tensor in this form.

We call this tree-based structure *compressed sparse fiber* (CSF). A tensor stored in this structure is called a CSF tensor. Leaves in a CSF tensor represent nonzeros, and sibling leaves form the fiber that is the result of holding all but the last index constant. These nonzero fibers are the key to fast MTTKRP algorithms.

Compression in a CSF tensor is visualized by the branching factor. Nodes with more than one child are the result of joining two or more repeated indices. Storage savings compound as we move deeper into a node's subtrees. For example, in Figure 1, the mode-1 tree uses only two nodes to store the mode-1 coordinates for all seven nonzeros.

When forming a CSF tensor, we follow an optimization used by SPLATT and first sort the tensor modes by increasing length. Using the smallest tensor mode as root nodes attempts to maximize the branching factor of the tree, resulting in additional compression.

4.2 MTTKRP Algorithms for CSF

Storing m different compressed tensors makes it easy to extract coarse-grained parallelism, but is not memory-efficient and does not scale to tensors with more than a few modes. Later, in Section 6.1 we show that SPLATT on average uses 80% more memory than the uncompressed data to store three compressed tensors. We will now detail how it is possible to perform MTTKRP on any mode of a CSF tensor while reducing floating-point operations in a manner similar to SPLATT.

4.2.1 Computing for the Root Mode

Computing for the mode stored at the root of a CSF tensor is a direct adaptation of the MTTKRP algorithm used by SPLATT. The keys of the root nodes determine which row of $\hat{\mathbf{A}}_1$ to update. We call our adaptation CSF-ROOT.

Nonzero $\boldsymbol{\mathcal{X}}(i,j,\ldots,l)$ scales the row vector $\mathbf{A}_2(j,:) * \cdots * \mathbf{A}_m(l,:)$. A sibling nonzero, $\boldsymbol{\mathcal{X}}(i,j,\ldots,p)$, will likewise require $\mathbf{A}_2(j,:) * \cdots * \mathbf{A}_m(p,:)$. Note that the multiplications only differ in the $\mathbf{A}_m$ terms which correspond to leaf nodes

Algorithm 2 MTTKRP on root nodes

```
1: function CSF-ROOT(node)
2:     d ← DEPTH(node)
3:     id ← IDX(node)          ▷ Extract the dth coordinate
4:     if d = m then
5:         ▷ Scale by the nonzero value if on the last level
6:         return VAL(node) · A_d(id, :)
7:     end if
8:
9:     Z(d, :) ← 0
10:    for c ∈ children(node) do
11:        Z(d, :) ← Z(d, :) + CSF-ROOT(c)
12:    end for
13:    return Z(d, :) ∗ A_d(id, :)
14: end function
```

in the tree. CSF-ROOT saves operations by factoring out the repeated multiplications and instead accumulates sibling nonzeros into a buffer. Once completed, the buffer is scaled by the factored multiplication and the result is propagated up the tree. The optimization is not unique to the leaf level; we can save operations by factoring redundant multiplications at each level of the tree.

Algorithm 2 shows a recursive implementation of CSF-ROOT. CSF-ROOT performs a depth-first traversal on the CSF tensor. Row $\hat{\mathbf{A}}_1(i,:)$ is completed at the end of the traversal of the ith tree. Hadamard products are efficiently accumulated in $\mathbf{Z}$, an $m \times F$ matrix which in general is a small constant in size. A node in the m_ith level accumulates the contributions from all children into $\mathbf{Z}(m_i,:)$ before propagating its own contributions up the tree. After all nodes in the ith tree have been processed, $\mathbf{Z}(1,:)$ is written to $\hat{\mathbf{A}}_1(i,:)$.

4.2.2 Computing for the Leaf Mode

MTTKRP on the longest mode uses the keys of the leaves to determine the output row of $\hat{\mathbf{A}}_m$. We use the same principle from CSF-ROOT to factor redundant Hadamard products and save floating-point operations. In this case, instead of accumulating nonzeros in a buffer to propagate up the tree, we recursively propagate Hadamard products down the tree and allow siblings to reuse previous multiplications. We call this algorithm CSF-LEAF and present a recursive implementation in Algorithm 3.

4.2.3 Computing for Interior Modes

Processing internal nodes relies on a combination of the CSF-ROOT and CSF-LEAF algorithms. Whereas CSF-ROOT determines write locations by the root nodes and CSF-LEAF uses the leaves, we now use one of the middle modes. We must process both parent and children subtrees before results can be stored. A recursive implementation of this algorithm, called CSF-INTERNAL, is detailed in Algorithm 4.

Assume we are computing for the wth level of the tree. We first propagate Hadamard products down the tree using CSF-LEAF until the we are $(w-1)$ levels deep. Next, we use CSF-ROOT to propagate the contributions of the entire node's subtree up to $\mathbf{Z}(w,:)$. The final result is the Hadamard product of $\mathbf{Z}(w-1,:)$ and $\mathbf{Z}(w,:)$.

Algorithm 3 MTTKRP on leaf nodes

```
1: function CSF-LEAF(node)
2:     d ← DEPTH(node)
3:     id ← IDX(node)          ▷ Extract the dth coordinate
4:     if d = m then
5:         ▷ Last level uses nonzeros and writes to Â_m
6:         Â_m(id, :) ← Â_m(id, :) + VAL(node) · Z(d, :)
7:         return
8:     end if
9:
10:    if d = 1 then                          ▷ Initialize Z
11:        Z(1, :) ← A_1(id, :)
12:    else
13:        ▷ Propagate Hadamard products down the tree
14:        Z(d, :) ← Z(d − 1, :) ∗ A_d(id, :)
15:    end if
16:    for c ∈ children(node) do
17:        CSF-LEAF(c)
18:    end for
19: end function
```

Algorithm 4 MTTKRP on mode w, an interior level

```
1: function CSF-INTERNAL(node)
2:     d ← DEPTH(node)
3:     id ← IDX(node)          ▷ Extract the dth coordinate
4:     if d = 1 then                ▷ Initialize the first level
5:         Z(1, :) ← A_d(id, :)
6:     else if d < w then       ▷ Move down to level w − 1
7:         Z(d, :) ← Z(d − 1, :) ∗ A_d(id, :)
8:         for c ∈ children(node) do
9:             CSF-INTERNAL(c)
10:        end for
11:    else   ▷ Use CSF-ROOT to complete computation
12:        Z(d, :) ← CSF-ROOT(node)
13:        Â_w(id, :) ← Â_w(id, :) + (Z(d − 1, :) ∗ Z(d, :))
14:    end if
15: end function
```

4.3 Parallel Formulation and Tiling

A parallelization of CSF-ROOT can exploit the same coarse-grained parallelism that SPLATT features. There are I_1 root nodes and each writes to a unique row of $\hat{\mathbf{A}}_1$. Thus, we can simply distribute whole trees to threads and avoid race conditions.

When parallelizing CSF-ROOT and CSF-LEAF, an important observation is that it is no longer safe to simply parallelize over trees and avoid race conditions. There is no guarantee that non-root keys are unique to the any tree, and thus we must now synchronize updates to the output matrix. One way of doing this is to reduce contention by storing a large number of mutexes, N, and using mutex $i \pmod N$ when writing to row i. This, however, is not an adequate solution. Even if we completely eliminate contention, at the leaf level we pay the price of locking and unlocking a mutex for every nonzero.

Instead, we developed a method that eliminates the need for locks by imposing an m-dimensional tiling over $\boldsymbol{\mathcal{X}}$. Suppose we are computing on the second mode of a tensor. We use the tiling along the second mode to induce a 1D partitioning

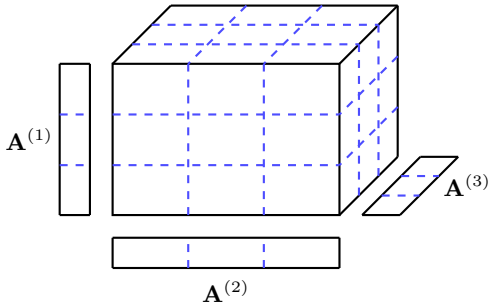


Figure 2: An m-dimensional tiling over $\boldsymbol{\mathcal{X}}$ for three processors. The tiles in whichever mode we are computing on form a 1D decomposition of the tensor. The tiling induces a partitioning of the matrix factors, which are distributed among threads to remove race conditions.

of the tensor nonzeros and $\hat{\mathbf{A}}_2$. If the 1D blocks of nonzeros are distributed to threads, due to the nature of the tiling they will write to non-overlapping rows of $\hat{\mathbf{A}}_2$. Figure 2 illustrates our tiling procedure for three threads on a three-mode tensor.

Lock-free parallelism comes with a price. Fibers that cross tile boundaries are treated as separate fibers, which increases storage overhead and decreases the number of factored multiplications. It is tempting to use a large number of small tiles in order to improve cache locality, but in practice this results in more storage than the uncompressed tensor and little to no performance benefit. Thus, we only use as many tiles per mode as there are available threads.

5. EXPERIMENTAL METHODOLOGY

5.1 Experimental Setup

The CSF data structure and associated algorithms were implemented and integrated into the the SPLATT toolkit. SPLATT is written in C using OpenMP with double-precision floating-point numbers and 64-bit integers. All source code is available for download[1]. Source code was compiled with GCC 4.9.2 using optimization level three.

During our discussion, we will refer to experiments on untiled tensors as CSF-M (CSF with mutexes) and tiled tensors as CSF-T (CSF with tiling). We benchmarked our CSF-based algorithms against two competing algorithms. COORD is a direct implementation of Equation (1) and is to our knowledge the fastest MTTKRP algorithm for uncompressed tensors. We use the original SPLATT kernel as a benchmark for compressed tensors.

We used $F = 16$ for all experiments. Experiments were performed on an HP ProLiant BL280c G6 blade server with two oct-core E5-2670 Xeon processors running at 2.6 GHz.

5.2 Datasets

Table 1 is a summary of the datasets we used for evaluation. The Netflix dataset is a *user-item-time* ratings tensor taken from the Netflix Prize competition [3]. Two datasets come from the Never Ending Language-Learning (NELL) project [4] which is freely available. Both tensors represent *noun-verb-noun* triplets. NELL-1 is a complete, extremely sparse tensor and NELL-2 is a smaller, more dense version in which the infrequent items have been pruned. Delicious is a *user-item-tag* dataset originally crawled by Görlitz et al. [7]. Beer [14] and Amazon [13] are *user-item-word* tensors parsed from Beer Advocate and Amazon product reviews, respectively. We used Porter stemming [15] on review text during parsing.

[1] `http://cs.umn.edu/~splatt/`

Table 1: Summary of datasets.

Dataset	$\mathbf{I}_1$	$\mathbf{I}_2$	$\mathbf{I}_3$	nnz
NELL-2	12K	9K	28K	77M
Beer	33K	66K	960K	94M
Netflix	480K	18K	2K	100M
Delicious	532K	17M	3M	140M
NELL-1	3M	2M	25M	143M
Amazon	5M	18M	2M	1.7B

nnz is the number of nonzero entries in the dataset. **K**, **M**, and **B** stand for thousand, million, and billion, respectively.

6. RESULTS

6.1 Comparison of Storage Requirements

Figure 3 shows the storage required for the uncompressed (COORD) tensor and the compressed tensor storage formats. CSF-M consistently has the smallest memory footprint and averages 68% less memory than SPLATT and 42% less memory than COORD. CSF-M achieves its best compression on NELL-1, with CSF using 73% less memory than SPLATT. After adding storage overhead from tiling for 16 threads, CSF-T still uses 58% less memory than SPLATT and 24% less than COORD. Overall, CSF-M and CSF-T always use less memory than COORD and SPLATT.

6.2 Performance Benchmarks

We evaluate the performance of our parallel CSF-based algorithms against SPLATT and COORD. We first examine performance on individual modes using the three parallel algorithms and compare against SPLATT and COORD timings on the same modes. We then examine total MTTKRP runtime.

Table 2 compares the runtimes of performing MTTKRP on the smallest tensor modes (the root nodes). Since CSF-ROOT is the SPLATT algorithm adapted to the CSF data structure, performance between the two methods is nearly identical. With the exception of the Netflix dataset, CSF-ROOT sees little benefit from tiling. Since tiles must be large, they offer little cache benefit and do not mitigate any locks. Netflix is an exception because its root and internal dimensions are very small compared to the leaf dimension. Most of the computation is done during nonzero accumulation instead of propagating results back up the tree.

Table 3 shows the runtimes of CSF-INTERNAL, which computes MTTKRP on the middle tensor modes (interior nodes). CSF-M outperforms SPLATT on half of the datasets. CSF-M and CSF-T both outperform SPLATT on the Ama-

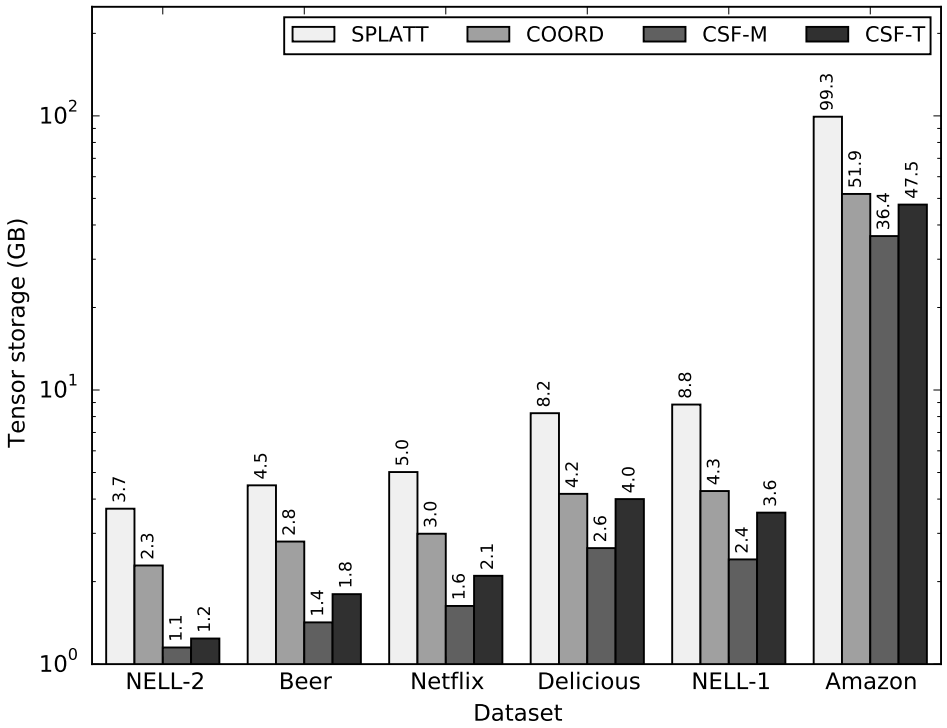


Figure 3: Required storage in gigabytes for each dataset, logarithmic scale.

Table 2: CSF-ROOT with 16 threads.

Dataset	COORD	SPLATT	CSF-M	CSF-T
NELL-2	4.30	0.10	0.10	0.11
Beer	5.20	0.12	0.12	0.17
Netflix	15.46	0.23	0.24	0.18
Delicious	11.25	0.55	0.56	1.22
NELL-1	18.95	1.11	1.00	1.29
Amazon	96.04	3.61	3.71	6.18

Values are the best time in seconds to execute MTTKRP on the smallest mode of the tensor. COORD is serial while SPLATT, CSF-M, and CSF-T all use 16 threads.

Table 3: CSF-INTERNAL with 16 threads.

Dataset	COORD	SPLATT	CSF-M	CSF-T
NELL-2	4.35	0.08	0.15	0.11
Beer	5.24	0.12	0.17	0.19
Netflix	15.55	0.50	0.33	0.23
Delicious	15.66	1.15	0.96	1.27
NELL-1	24.22	1.10	1.18	1.73
Amazon	97.12	15.91	12.28	6.96

Values are the best time in seconds to execute MTTKRP on the interior mode of the tensor. COORD is serial while SPLATT, CSF-M, and CSF-T all use 16 threads.

zon tensor with 1.7 billion nonzeros, with CSF-T reaching 2.3× speedup over parallel SPLATT. We attribute this to CSF-based algorithms sometimes performing fewer FLOPs than SPLATT on non-root modes. This is possible because different views of the same tensor can results in drastic differences in computation. For example, a *user-item-time* ratings tensor will have more nonzeros in the (*item, time*) fibers than (*user, item*) because users are unlikely to rate items more than once. The discrepancy between mode representations is illustrated in Figure 1.

Tiling improves performance of CSF-INTERNAL on three tensors. We only have to lock when writing to an internal node, and thus if there is a large amount of computation in the leaves of the tree it may be more beneficial to skip tiling.

CSF-LEAF runtimes are shown in Table 4. SPLATT is faster than CSF-M on all but the smallest tensor, NELL-2. Leaf computation operates on the longest mode of the tensor and is the most taxing MTTKRP step on the CPU memory hierarchy because each nonzero can result in a write to distant memory outside of the CPU cache. As expected, parallel scalability is very poor without tiling because a mutex is locked and unlocked on every nonzero.

Finally, we examine the total MTTKRP runtime in Table 5. CSF-M averages 40% as fast as SPLATT without tiling and 81% as fast with tiling enabled. CSF-T is comparable in performance to SPLATT even on our largest tensor, while using over 50 gigabytes less memory.

7. CONCLUSIONS AND FUTURE WORK

Tensor factorization has emerged as a growing tool for large-scale data analysis. Modern tensors are extremely sparse and feature irregular data accesses as a result of their mode-centric computations. In this work, we presented the compressed sparse fiber (CSF) format for sparse tensors and three associated shared-memory parallel algorithms for performing tensor-matrix multiplication. This is the first parallel work to only use a single compressed tensor representa-

Table 4: CSF-LEAF with 16 threads.

Dataset	COORD	SPLATT	CSF-M	CSF-T
NELL-2	4.49	0.10	1.21	0.10
Beer	5.23	0.17	1.71	0.21
Netflix	15.65	0.15	1.51	0.40
Delicious	15.06	1.07	2.43	2.30
NELL-1	18.44	1.58	3.16	3.01
Amazon	96.47	3.79	121.05	11.07

Values are the best time in seconds to execute MTTKRP on the longest mode of the tensor. COORD is serial while SPLATT, CSF-M, and CSF-T all use 16 threads.

Table 5: Total MTTKRP iteration time with 16 threads.

Dataset	COORD	SPLATT	CSF-M	CSF-T
NELL-2	13.14	0.28	1.46	0.31
Beer	15.67	0.41	2.00	0.57
Netflix	46.66	0.89	2.08	0.81
Delicious	41.97	2.77	3.95	4.79
NELL-1	61.60	3.79	5.34	6.03
Amazon	289.64	23.31	137.03	24.21

Values are the best time in seconds to execute MTTKRP across all three modes. COORD is serial while SPLATT, CSF-M, and CSF-T all use 16 threads.

tion that allows for significant memory savings and consistent data access patterns. On average, our algorithms used 58% less memory than the state-of-the-art while achieving 81% of its parallel performance.

There are several points of future work. One direction is to investigate the cache-friendly reorderings developed in [19]. It may be possible to exploit the single access pattern of the tensor and find even more successful reorderings than before. Second, it may be beneficial to only tile over some modes of the tensor. The mixed results of CSF-INTERNAL suggest that is sometimes better to use locks instead of introducing the overhead of tiling for parallelism. Additionally, this approach would lessen the memory overhead that results from tiling, allowing for work with larger multi-core systems.

Acknowledgments

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), 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.

8. REFERENCES

[1] B. W. Bader and T. G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, December 2007.

[2] B. W. Bader, T. G. Kolda, et al. Matlab tensor toolbox version 2.6, Feb. 2015. http://www.sandia.gov/~tgkolda/TensorToolbox/.

[3] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[4] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka, and T. M. Mitchell. Toward an architecture for never-ending language learning. In *In AAAI*, 2010.

[5] J. D. Carroll and J.-J. Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart-Young" decomposition. *Psychometrika*, 35(3):283–319, 1970.

[6] J. H. Choi and S. Vishwanathan. DFacTo: Distributed factorization of tensors. In *Advances in Neural Information Processing Systems*, pages 1296–1304, 2014.

[7] O. Görlitz, S. Sizov, and S. Staab. Pints: peer-to-peer infrastructure for tagging systems. In *IPTPS*, page 19, 2008.

[8] J. C. Ho, J. Ghosh, and J. Sun. Marble: high-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 115–124. ACM, 2014.

[9] O. Kaya and B. Uçar. Scalable sparse tensor decompositions in distributed memory systems. Technical report, 2015.

[10] T. G. Kolda and B. Bader. The TOPHITS model for higher-order web link analysis. In *Proceedings of Link Analysis, Counterterrorism and Security 2006*, 2006.

[11] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[12] S. Leurgans and R. T. Ross. Multilinear models: applications in spectroscopy. *Statistical Science*, pages 289–310, 1992.

[13] J. McAuley and J. Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 165–172. ACM, 2013.

[14] J. McAuley, J. Leskovec, and D. Jurafsky. Learning attitudes and attributes from multi-aspect reviews. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 1020–1025. IEEE, 2012.

[15] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[16] N. Ravindran, N. D. Sidiropoulos, S. Smith, and G. Karypis. Memory-efficient parallel computation of tensor and matrix products for big tensor decomposition. In *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, 2014.

[17] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, A. Hanjalic, and N. Oliver. 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*, pages 155–164. ACM, 2012.

[18] S. Smith and G. Karypis. Dms: Distributed sparse tensor factorization with alternating least squares. Technical report, 2015.

[19] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *International Parallel & Distributed Processing Symposium (IPDPS'15)*, 2015.