An Investigation of Sparse Tensor Formats for Tensor Libraries

by

Parker Allen Tew

S.B., Massachusetts Institute of Technology (2015)

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

Author . . . .
Department of Electrical Engineering and Computer Science
July 29, 2016

Certified by . . . .
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

An Investigation of Sparse Tensor Formats for Tensor Libraries

by

Parker Allen Tew

Submitted to the
Department of Electrical Engineering and Computer Science
on July 29, 2016
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Tensors provide a generalized structure to store arbitrary indexable data, which is applicable in fields such as chemometrics, physics simulations, signal processing and lies at the heart of machine learning. Many naturally occurring tensors are considered sparse as they contain mostly zero values. As with sparse matrices, various techniques can be employed to more efficiently store and compute on these sparse tensors.

This work explores several sparse tensor formats while ultimately evaluating two implementations; one based on explicitly storing coordinates and one that compresses these coordinates. The two formats, Coordinate and CSF2, were evaluated by comparing their execution time of tensor-matrix products and the MTTKRP operation on several datasets. We find that the Coordinate format is superior for uniformly distributed sparse tensors or when used in computation that emits a sparse tensor via a mode dependent operation. In all other considered cases for large sparse tensors, the storage savings of the compressed format provide the best results.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Contents

List of Figures

Acknowledgements

First, I thank my advisor, Saman Amarasinghe, and Fredrik Kjolstad for their help throughout my research. Saman gave me the opportunity to work on extremely challenging problems and allowed me to grow as a computer scientist. Fred was a guiding hand and mind throughout my research. His willingness to listen and give advice made my research possible. Also, a special thanks to the rest of the Commit group at MIT for their influence on my work.

Most importantly, I thank my parents, Frank and Monica, and family for their unwavering support throughout my life. They provided me with love and care through the highs and lows of MIT. Lastly, thanks to my dog, Tate, for reminding me to keep life simple.

Chapter 1

Introduction

Fields ranging from economics to computer graphics use matrix-based linear algebraic operations to solve some of their computational problems. Multi-linear algebra, a generalization of linear algebra, provides a language for operating on tenors, which are multi-dimensional array structures. The ability to store higher dimensional data has been used in various fields such as chemometrics, physics simulations, signal processing, computer vision and machine learning [1]. As machine learning has become more and more popular in academia and industry, there is a growing need for an efficient standardized tensor library that unlocks the power of tensors. This work sheds light on how such a library should internally store and operate on tensors.

Tensors provide a generalization of storing indexed data. They allow for an arbitrary number of dimensions, where each dimension is termed a mode. Commonly used scalars, vectors and matrices are all specific types of tensors existing in different number of dimensions. In turn, they each require a different number of index variables to access their underlying data. Scalars exist without any index variables rendering them dimensionless and are called zeroth-order tensors. Vectors require a single index variable, have one mode and are termed first-order tensors. Matrices require two index variables, have two modes and are called second-order tensors. By generalizing to higher dimensions, tensors provide a powerful abstraction that can store and relate data for an arbitrary number of index variables.

As seen with matrices, not all combinations of the index values have meaningful data. For example, recommender systems operating on movie reviews might want to relate movies, users, and review dates to a user rating in order to produce new movie recommendations. Here there are three index variables that relate to a rating; movies, users, and dates. As is often the case with services like Netflix, it is a safe assumption that most users have not rated every movie in the Netflix catalog. This sparse relationship between users and movies results in a sparse tensor, where the majority of the data values, or in this case ratings, are zero. Because it often only takes one sparse dimension to make an entire tensor sparse, many naturally occurring tensor datasets are in fact sparse. By only storing and operating on the non-zero values, not only can the storage requirements of a tensor be reduced, but sparsely stored tensors also eliminate unnecessary computations involving zeros.

This work explores various formats for storing and operating on sparse tensors. We find that the Coordinate format is superior for uniformly distributed sparse tensors or when used in computations that emit a sparse tensor via a mode dependent operation. In all other considered cases for large sparse tensors, the storage savings of the compressed format provide the best results.

With this information, tensor libraries can make an informed decision on how they represent sparse tensors. By choosing the most efficient format, users may operate on larger and more complex problems. This work contributes to future development of tensor libraries by (1) providing a fair comparison and analysis of multithreaded implementations of Coordinate and Compressed Sparse Fiber formats, (2) developing a technique to efficiently partition Coordinate structures for parallel implementations using a *slices* array, and (3) suggesting guiding rules on how future libraries could benefit from these formats most efficiently.

Chapter 2

Background

In this chapter we present the necessary background information to talk about tensors, sparse tensors formats, and a few tensor operations. For information regarding additional tensor operations, please refer to a sparse tensor survey by Bader and Kolda [2].

2.1 Notation

The notation for tensors is based on a commonly used matrix notation that can be extended to general tensors. As matrices have two modes, their rows and columns, accessing matrix data requires two index variables: i and j. The notation for accessing matrix data is $A(i, j)$. Rows of a matrix can be accessed by $A(i, :)$, where the row index variable, i, is kept fixed and the column index variable, j, ranges across the column dimension. Similarly, $A(:, j)$ accesses columns by allowing i to vary while the j is held constant.

Tensors can consist of an arbitrary number of dimensions thus requiring an arbitrary number of index variables. This work will focus on third-order tensors, which only require three index variables i, j, and k that span $\mathbb{R}^{I \times J \times K}$. Tensor data can be access via $\mathbf{X}(i, j, k)$ for the index values i, j, and k. A third-order tensor is represented in Figure 2.1.

As with matrices, we can select ranges of a tensor by fixing some index variables and varying others. Although this process can be used to select any sub-tensor within a nth-order tensor, we focus on how it used for third-order tensors. Matrix slices of a

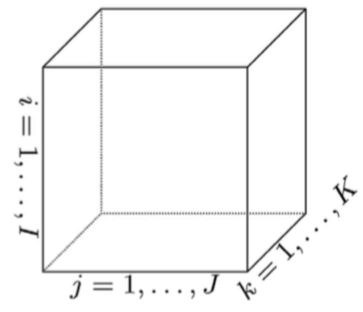


Figure 2.1: A third-order tensor: $\mathbf{X} \in \mathbb{R}^{I \times J \times K}$ [3].

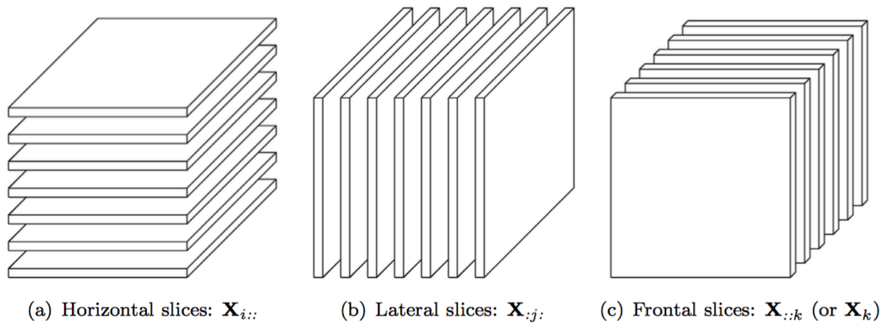
third-order tensor can be attained by fixing one index variable and varying the other two. When two index variables are kept fixed, we get vectors. When referring to vectors as sub-tensors, we call them tensor fibers. For example, the rows and columns of a matrix are tensor fibers. A general rule is that the number of free index variables defines the dimensionality of the sub-tensor being accessed. These techniques are laid out in 2.2 and 2.3.

Some tensor operations require specifying a mode of operation. Viewing the tensor as a series of slices or grid of fibers gives us an easier way to conceptualize these operations. For example, when we look at a mode-i fiber view of $\mathbf{X} \in \mathbb{R}^{I \times J \times K}$, the first index variable i is free to vary while the other index variables are kept constant. This results in fiber columns. This process can be performed for any mode to produce fibers along any axis. Fiber rows are achieved by allowing the j index variable to vary while fixing i and k. For the last mode, fiber tubes are produced when k is the only index variable that is free to vary. Again, this is all shown in Figure 2.3.

Another key tensor operation is tensor matricization, which produces a matrix from a tensor by flattening the tensor to two dimensions. This is achieved conceptually by concatenating the matrix slices of the tensor. Matricization is mode dependent and the shape of the resulting matrix depends on the specified mode. For example, a mode-i matricization is denoted by $\mathbf{X}_{(1)} \in \mathbb{R}^{I \times JK}$. Figure 2.4 shows this process for all three modes of a $2 \times 3 \times 2$ tensor.

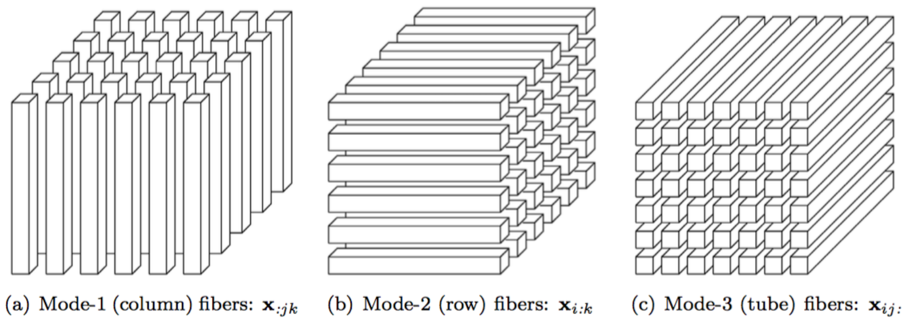
2.2 Matrix Operations

Tensor decompositions discussed later in this chapter rely on several specific matrix operations. The first is the Hadamard operation, which is an element-wise multiplication



(a) Horizontal slices: $\mathbf{X}_{i::}$
(b) Lateral slices: $\mathbf{X}_{:j:}$
(c) Frontal slices: $\mathbf{X}_{::k}$ (or $\mathbf{X}_k$)

Figure 2.2: The different mode dependent slices of $\mathbf{X} \in \mathbb{R}^{I\times J\times K}$ [3].



(a) Mode-1 (column) fibers: $\mathbf{x}_{:jk}$
(b) Mode-2 (row) fibers: $\mathbf{x}_{i:k}$
(c) Mode-3 (tube) fibers: $\mathbf{x}_{ij:}$

Figure 2.3: The different mode dependent fibers of $\mathbf{X} \in \mathbb{R}^{I\times J\times K}$ [3].

$$\boldsymbol{\mathcal{X}}(:,:,1) = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \boldsymbol{\mathcal{X}}(:,:,2) = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

$$\mathbf{X}_{(1)} = \begin{bmatrix} 1 & 2 & 3 & 7 & 8 & 9 \\ 4 & 5 & 6 & 10 & 11 & 12 \end{bmatrix}$$

$$\mathbf{X}_{(2)} = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

$$\mathbf{X}_{(3)} = \begin{bmatrix} 1 & 4 & 2 & 5 & 3 & 6 \\ 7 & 10 & 8 & 11 & 9 & 12 \end{bmatrix}$$

Figure 2.4: This figure shows the matricizations of a $2 \times 3 \times 2$ tensor. The first row defines the tensor with two matrix slices. $X_{(1)}$, $X_{(2)}$ and $X_{(3)}$ show the different possible matricizations. [4]

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} A_{11}\mathbf{B} & A_{12}\mathbf{B} & \cdots & A_{1n}\mathbf{B} \\ A_{21}\mathbf{B} & A_{22}\mathbf{B} & \cdots & A_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1}\mathbf{B} & A_{m2}\mathbf{B} & \cdots & A_{mn}\mathbf{B} \end{pmatrix}.$$

Figure 2.5: A Kronecker product resulting in many scaled copies of input matrix, **B**.

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \qquad \mathbf{C} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix} \qquad \mathbf{C} \odot \mathbf{B} = \begin{bmatrix} c_{11}b_{11} & c_{12}b_{12} \\ c_{11}b_{21} & c_{12}b_{22} \\ c_{21}b_{11} & c_{22}b_{12} \\ c_{21}b_{21} & c_{22}b_{22} \\ c_{31}b_{11} & c_{32}b_{12} \\ c_{31}b_{21} & c_{32}b_{22} \end{bmatrix}$$

Figure 2.6: A Khatri-Rao product between a 3×2 matrix and 2×2 matrix. The output is a 6×2 matrix [4].

of two matrices, $\mathbf{A} \in \mathrm{I\!R}^{I \times J}$ and $\mathbf{B} \in \mathrm{I\!R}^{I \times J}$, that result in $\mathbf{C} \in \mathrm{I\!R}^{I \times J}$. $\mathbf{A}(i, j)$ is scaled by $\mathbf{B}(i, j)$ and stored at $\mathbf{C}(i, j)$. The Hadamard operation is denoted by $\mathbf{A} * \mathbf{B}$ and is a fundamental operation in a Kronecker product.

The Kronecker product between $\mathbf{A} \in \mathrm{I\!R}^{M \times N}$ and $\mathbf{B} \in \mathrm{I\!R}^{P \times Q}$ results in $\mathbf{C} \in \mathrm{I\!R}^{MP \times NQ}$ and is represented by $\mathbf{A} \otimes \mathbf{B}$. Conceptually, every element in **A** scales the entire matrix **B** and is stored at $\mathbf{C}(m, n)$. Each row of the output matrix stores N scaled versions of **B** and each column stores M scaled **B** matrices. This structure gives the output matrix **C** its dimensions of $MP \times NQ$. An example of the Kronecker product is shown in Figure 2.5.

Tensor decompositions, described in more detail in the next section, perform many Khatri-Rao products, which are column-wise Kronecker products. The Khatri-Rao product is denoted by $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$, where $\mathbf{A} \in \mathrm{I\!R}^{I \times J}$, $\mathbf{B} \in \mathrm{I\!R}^{M \times J}$ and $\mathbf{C} \in \mathrm{I\!R}^{IM \times J}$. Figure 2.6 gives a clearer view of a Khatri-Rao product.

2.3 Tensor Operations

Using the matrix operations defined above, we are ready to introduce several tensor operations. There are numerous tensor operations that are not considered here. In this work, we focus our discussion and evaluation on Tensor-Matrix products and Tensor Decompositions.

2.3.1 Tensor-Matrix Products

Tensor-Matrix products conceptually consist of slicing an input tensor into a set of matrices and performing a series of matrix-matrix products with the input matrix. When the matrix products are recombined and assuming the input tensor is sparse, the result is a sparse tensor. As shown in Figure 2.2, slicing is mode dependent and determines the shape of the output tensor. Consider inputs $\mathbf{X} \in \mathbb{R}^{I \times J \times K}$ and $\mathbf{A} \in \mathbb{R}^{R \times J}$ performing a mode-j Tensor-Matrix product denoted by $\mathbf{Y}{=}\mathbf{X}{\times_2}\mathbf{A} \in \mathbb{R}^{I \times R \times K}$. This operation can be formalized by:

$$\mathbf{Y}(i,r,k) = \sum_j \mathbf{X}(i,j,k)\mathbf{A}(r,j)$$

Inputs $\mathbf{X}$ and $\mathbf{A}$ share index variable j while all other index variables remain free. Notice that the dimension of the mode of operation of $\mathbf{X}$ must match the number of columns in $\mathbf{A}$. Additionally, $\mathbf{Y}$ has the same dimensions as $\mathbf{X}$ except the dimension of the mode of operation is now the number of rows in $\mathbf{A}$, instead of the number of columns.

Tensor-Matrix products acts as a way to transform old dimensions into new ones. One hypothetical use case of a Tensor-Matrix product would be transforming $\mathbf{X}$ with dimensions movie, user, review date to dimensions director, user, review date. This is achieved by $\mathbf{X} \times_1 \mathbf{A}$, where $\mathbf{A}$ has dimensions directors and movies. As directors generally direct multiple movies, the movie dimension will be contracted and replaced with the smaller director dimension resulting in a tensor with dimensions of director, user, and review date.

For simplification of software and analysis, we are assuming input matrices to be dense, which causes the replaced dimension of the output tensor to also become dense. In practice, these input matrices can very well be sparse. It simply depends on the data. Yet when using a dense matrix in practice, if the dimension being replaced is sparse, the number of rows of the input matrix usually needs to be much smaller than the replaced dimension. Otherwise, the amount of storage required the output tensor blows up in size. For a uniformly distributed tensor, where there are few values per fiber, the number of non-zero values could increase by a factor close to the number of rows of the input matrix.

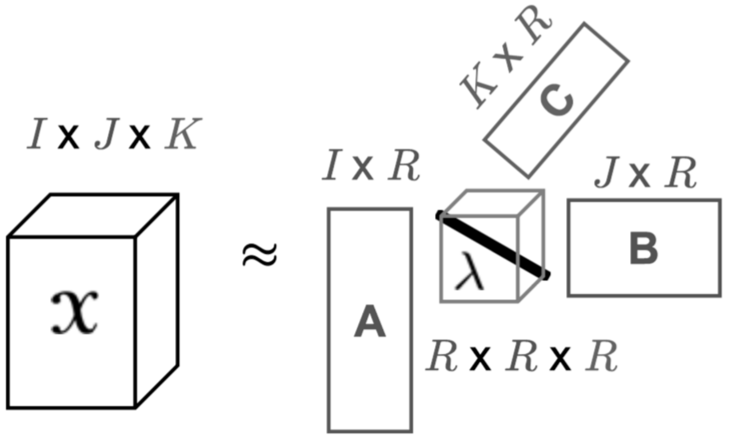


Figure 2.7: A CPD of $\mathbf{X} \in \mathbb{R}^{I \times J \times K}$ into $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$ and $\lambda \in \mathbb{R}^{R \times R \times R}$. Here the factor matrices have been normalized and the weights are put into λ, a diagonal tensor. TODO-CITATION

2.3.2 Tensor Decompositions

The goal of a tensor decomposition is to deconstruct the original tensor into a series of components that give a distilled view of the tensor's characteristics. Some of the information easily gleaned from a decomposed tensor is often lost in the complexity of the full tensor. Tensor decompositions provide the same functionality as matrix factorization does for matrices. We focus on the Canonical Polyadic Decomposition (CPD), sometimes referred to as PARAFAC, which is a popular decomposition technique for arbitrary tensors [5]. The CPD is a generalization of singular value decomposition (SVD) used for matrices that often uses alternating least squares (ALS) to optimize its output.

While the CPD is useful for arbitrary tensor decompositions, we will focus on how it is applied to third-order tensors. Using the CPD, the tensor $\mathbf{X} \in \mathbb{R}^{I \times J \times K}$ is decomposed into three matrices: $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$ and $\mathbf{C} \in \mathbb{R}^{K \times R}$. Sometimes the factor matrices are all normalized and the weights are absorbed into a scaling vector or diagonal tensor. The CPD with the small factored scaling tensor is represented in Figure 2.7. The following expressions produce approximate matricizations of the original tensor:

$$\mathbf{X}_{(i)} \approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^\intercal,\ \mathbf{X}_{(j)} \approx \mathbf{B}(\mathbf{C} \odot \mathbf{A})^\intercal,\ \mathbf{X}_{(k)} \approx \mathbf{C}(\mathbf{B} \odot \mathbf{A})^\intercal$$

As stated previously, the CPD often uses ALS to solve the non-convex optimization problem:

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} ||\mathbf{X}_{(i)} - \mathbf{A}(\mathbf{C} \odot \mathbf{B})^\intercal||_R.$$

During each iteration of CPD, each factor matrix is computed by keeping the other matrices fixed. The calculation of $\mathbf{A}$ for a single iteration is:

$$\mathbf{A} = \mathbf{X}_{(i)}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^{\intercal}\mathbf{C} * \mathbf{B}^{\intercal}\mathbf{B})^{\dagger}$$

, where † represents computing the pseudo-inverse of the matrix $(\mathbf{C}^{\intercal}\mathbf{C} * \mathbf{B}^{\intercal}\mathbf{B})$. Because $(\mathbf{C}^{\intercal}\mathbf{C} * \mathbf{B}^{\intercal}\mathbf{B})$ is a sufficiently small $R \times R$ matrix, it is relatively inexpensive to compute. Therefore, a majority of the work comes from computing the Matricized Tensor times Khatri-Rao product (MTTKRP), $\mathbf{X}_{(i)}(\mathbf{C} \odot \mathbf{B})$. Because of this, the MTTKRP operation is a good indicator of how well different sparse tensor formats will perform when computing a full CPD. Pseudo-code describing the CPD algorithm for arbitrary tensors is included in Algorithm 1. For more low-level details of the CPD, readers are invited to refer to the comprehensive report put forth by Bader and Kolda [3].

As the main purpose of the decomposition is to reduce the original tensor into a manageable number of important factors, the rank R of the decomposition is usually much smaller than the largest dimension of the original tensor. This ensures the resulting matrix factors are easier to interpret and also limits the computational requirements as smaller factor matrices require fewer operations for construction.

Algorithm 1 Canonical Polyadic Decomposition with ALS [6]

```
1: while not converged do
2:     for $i \in \{1, 2, \ldots, m\}$ do
3:         $\mathbf{M} \leftarrow \mathbf{1}^{R \times R}$
4:         for $j \in \{1, 2, \ldots, m\} \setminus \{i\}$ do
5:             $\mathbf{M} \leftarrow \mathbf{M} * (\mathbf{A}_j^T \mathbf{A}_j)$
6:         $\hat{\mathbf{A}}_i \leftarrow \mathbf{X}_i(\mathbf{A}_m \odot \cdots \odot \mathbf{A}_{i+1} \odot \mathbf{A}_i \odot \cdots \odot \mathbf{A}_1)$
7:         $\mathbf{A}_i \leftarrow \hat{\mathbf{A}}_i(\mathbf{M}^{-1})$
8:         Normalize columns of $\mathbf{A}_i$
```

2.4 Sparse Tensors

As previously stated, sparse tensors are found throughout real world applications. Tensors are considered sparse if they hold a majority of zero values. Sparse tensor storage formats allow us to only store non-zero values thereby reducing storage requirements and eliminating unnecessary silent computations involving zero values. This section discusses two sparse tensor formats: the Coordinate and Compressed Sparse Fiber formats.

2.4.1 Coordinate Format

The Coordinate format benefits from its simplicity and flexibility. A Coordinate stored sparse tensor consists of storing all non-zero values and all of their corresponding coordinates. The Coordinate structure does not compress the tensor, but reduces storage by removing all zero values. For example, in a third-order tensor, each non-zero values is accompanied by three index values for i, j and k. Figure 2.9(a) shows an example of a fourth-order tensor stored this way. Generally, the coordinate format requires $O(m{*}nnz)$ index values and $O(nnz)$ data values, where m is the number of modes of a tensor and nnz is the number of non-zero tensor values.

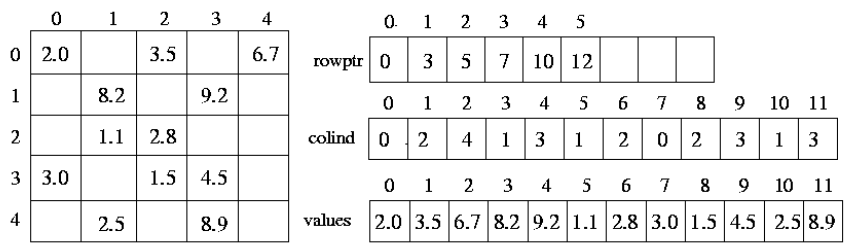
Some attractive features of Coordinate based tensors are they allow for $O(1)$ insertions for new non-zero elements into an unsorted tensor, but $O(log(nnz))$ for finding the insert location and potentially $O(n)$ for the insert while maintaining the sort. Coordinate tensors also allow for in-place sorting, which allows for relatively fast transformations of the coordinate structure for mode dependent operations. Additionally, Tensor-Matrix products output sparse tensors and building a Compressed Sparse Fiber tensor on the fly may be prohibitively expensive. This makes a coordinate like structure a necessity for any comprehensive sparse tensor library.

2.4.2 Compressed Sparse Fiber Format

Before we describe the compression of sparse tensors, we review the successful techniques used for compressing sparse matrices. These techniques are generalized for compressing tensors.

Review of Compressed Sparse Matrix Formats

Sparse matrices are often stored in either Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) used by Gustavson [7]. CSR involves three one-dimensional arrays. One of the arrays stores the non-zero values. Another array is dedicated to storing the column index values of each non-zero value. The final array is a row pointer array that stores indices into the column index array. Each index into the column array signifies the beginning of a new row. The row index value is implicitly stored as the position of the index in the row pointer array. The number of elements in a given row is determined by the difference between the current index into the column array and



	0	1	2	3	4
0	2.0		3.5		6.7
1		8.2		9.2	
2		1.1	2.8		
3	3.0		1.5	4.5	
4		2.5		8.9	

	0	1	2	3	4	5			
rowptr	0	3	5	7	10	12			

	0	1	2	3	4	5	6	7	8	9	10	11
colind	0	2	4	1	3	1	2	0	2	3	1	3

	0	1	2	3	4	5	6	7	8	9	10	11
values	2.0	3.5	6.7	8.2	9.2	1.1	2.8	3.0	1.5	4.5	2.5	8.9

Figure 2.8: A matrix stored in CSR format.

the next index in the column array. Figure 2.8 depicts a matrix stored in CSR. The Coordinate format requires $2 * nnz + I + 1$ storage, where nnz is the number of non-zero values and I is the number of rows.

CSC is essentially the same as CSR except the rows and columns are switched. These compression formats are beneficial as they reduce storage, but also lend themselves nicely towards row/column based operations as they inherently group row/column values together. This comes at the cost of expensive insertion and matrix transpose operations.

Generalizing CSR/CSC for Tensors

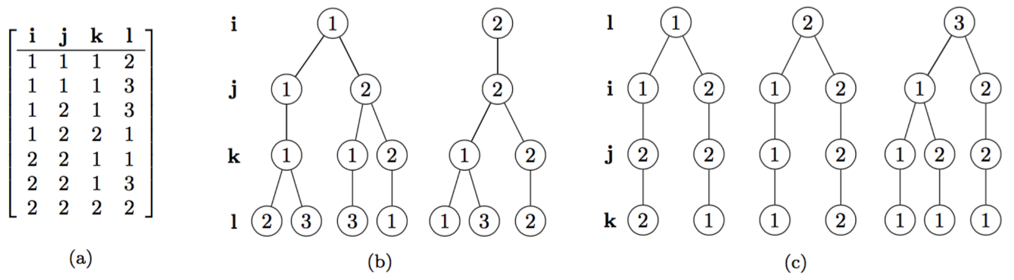
The general goal when compressing a sparse tensor is to reduce the duplication of index values that are shared by multiple non-zero values. There are several variations of compressed sparse tensor formats, but all are in one form or another based on the same principles as CSR/CSC. The most general sparse tensor format, Compressed Sparse Fiber (CSF), is introduced by Smith and Karypis [4]. CSF extends CSR/CSC by layering index pointer arrays to compress the additional tensor dimensions, thereby reducing the duplication of index values. This technique of layering can be thought of as building a tree-like representation of the tensor. Figure 2.9(b,c) illustrate the effects on this tree-like representation when using two different compression orderings on the same tensor. Any time a node is split into subtrees or leaf nodes, duplicate index values are eliminated. As matrices can choose between CSR/CSC to compress their two modes, higher-dimensional tensors have many more compression orderings. For a three-mode tensor with s slices, f fibers and nnz non-zero values, it requires $2 * (nnz + s + f) + 2$ values.

This technique of adding additional layers of index pointer arrays can continue for arbitrary dimensions. The amount of compression achieved relies on the compressibility of the tensor data as well as the compression order. Bader and Kolda express explicit concern about the decision of compression ordering stating that with N dimensions there are a possible $N!$ potential compression orderings [2]. For matrix compression, there are two dimensions and hence only two compression ordering options: rows or columns. Despite this concern, there are several simple compression ordering heuristics that prove useful in choosing the a good compression order.

To explore different compression ordering heuristics, we continue with the Netflix movie rating example which used a tensor composed of movies, users, and review dates. The sparsity of the tensor comes from the fact the average user rarely submits ratings and for few movies. However, there is surely to be at least several reviews from the large user base each day for multiple movies. Therefore, this dense day dimension makes the optimal compression order: review dates, movies and users. This equates to sorting each dimension by density and compressing the densest dimensions first. Conceptually this makes sense in relation to the tree view of the tensor because dense dimensions have the highest likelihood of repeating index values and therefore will be duplicated less often if they are higher up in the tree. To perform this heuristic, dimensions need to be sorted by the ratio of the number of their unique index values to their size, which might be potentially expensive. A cheaper heuristic could involve compressing smaller dimensions first as is suggested by Smith and Karypis [6]. Additionally, as shown with the Netflix example, the user can leverage their knowledge of the data set to provide a compression ordering, but this is a last resort as it is ideal to take away this burden from the user.

It is important to keep in mind that the saving on data storage isn't always the most important factor when choosing an ordering. The specific operations and modes of operations are likely the most important factors in determining how efficiently the tensor can be used.

Lastly, uniformly distributed sparse tensors can in fact require more storage when stored in a compressed format. If the non-zero values are distributed such that there are rarely multiple non-zero values in a given fiber, the compression overhead will outweigh the savings of any compression. Hence, uniformly distributed sparse tensors see no benefit from compression and should be stored in the Coordinate format.



i	j	k	l
1	1	1	2
1	1	1	3
1	2	1	3
1	2	2	1
2	2	1	1
2	2	1	3
2	2	2	2

Figure 2.9: Sparse tensor storage formats for a four-mode tensor. (a) Coordinate format (b) CSF format with $i{-}j{-}k{-}l$ compression (c) CSF format with $l{-}i{-}j{-}k$ compression. [6]

Chapter 3

Related Work

In this section, we mention the existing relevant work in the field of sparse tensors. These works were heavily influential in our research.

3.1 MATLAB Tensor Toolbox

MATLAB alone supports a variety of element-wise and binary dense tensor operations. Bader and Kolda introduced the MATLAB Tensor Toolbox, which extends MATLAB to support tensor multiplication and matricization [8]. However, Bader and Kolda also point out that MATLAB cannot support the storage of sparse tensors beyond a set of sparse matrices, which are stored in CSC [9].

In addition to providing details on their MATLAB Tensor Toolbox, Bader and Kolda also include a survey regarding tensor operations and sparse tensor formats [8]. When discussing the decision between the Coordinate and compressed tensor formats, Bader and Kolda suggest using a Coordinate based format for its simplicity. They identify that compressing a tensor with a specific compression ordering is less important as you add dimensions. They also point out that with a compressed format, a compression order must be specified. For N dimensions, this results in $N!$ unique orderings. This concern about compression order is addressed in Chapter 4 as we take a closer look at compressed sparse tensor formats. Readers curious about other tensor operations are encouraged to examine this comprehensive resource [8].

3.2 Extended Karnaugh Map Representation

Lin, Liu and Chung propose using the Extended Karnaugh Map Representation (EKMR) to compress matrices for third-order and fourth-order tensors [10]. This representation involves matricizing the tensor and then using CSR or CSC to compress it. Although it was found to be efficient for slice multiplication, the multiplication of the matrices of two different tensors, Bader and Kolda gave their opinion that this type of format was not a general solution for tensor libraries. However, a tensor library in a more mature state could leverage such a specialized format for specific use cases.

3.3 SPLATT and Compressed Sparse Fiber

Smith and Karypis first introduce SPLATT, a C based third-order tensor library that uses a compressed sparse tensor format [4]. SPLATT supports fast third-order tensor decomposition via a shared-memory parallelized MTTKRP implementation. For the MTTKRP operation, $\hat{\mathrm{A}}= X_{(1)}(C \odot B)$, Smith and Karypis point out that explicitly computing $(C \odot B)$, where $\mathbf{B} \in \mathrm{I\!R}^{J\times R}$ and $\mathbf{C} \in \mathrm{I\!R}^{K\times R}$, creates a dense $JK \times R$ matrix that can require much more memory than the original tensor. To avoid this, they derive an alternative algorithm for computing a MTTKRP that computes rows at a time and saves FLOPs. The derivation for this new algorithm is shown in Figure 3.1. Our work makes use of this alternative algorithm when evaluating MTTKRP with our formats. SPLATT compensates for the inflexibility of compressed tensor formats by storing a compressed tensor for each mode. This allows for faster mode dependent operations, like MTTKRP, at the cost $3x$ the storage footprint.

SPLATT is extended by Smith and Karypis with the introduction of the Compressed Sparse Fiber (CSF) format that allows operating on tensors of arbitrary dimensions [6]. CSF improves SPLATT in several impressive ways. It uses a single compressed tensor structure, thereby eliminating the need for storing multiple compressed versions. To do this, Smith and Karypis provide three new algorithms for CSF that can compute a MTTKRP regardless of the compression order of the tensor. Additionally, Smith and Karypis go on to provide a tiling scheme for CSF that greatly increases caching performance and the overall runtime of MTTKRP operations and by extension CPDs. We compare the performance of our implementation of the Coordinate and CSF2 formats

$$\begin{aligned}
\hat{\mathbf{A}}(i,r) &= \sum_{z=0}^{JK} \mathbf{X}_{(1)}(i,z)\mathbf{B}(z\%J,r)\mathbf{C}(z/J,r) \\
\hat{\mathbf{A}}(i,:) &= \sum_{z=0}^{JK} \mathbf{X}_{(1)}(i,z)(\mathbf{B}(z\%J,:) * \mathbf{C}(z/J,:)) \\
&= \sum_{k=0}^{K}\sum_{j=0}^{J} \mathbf{X}_{(1)}(i,j,k)(\mathbf{B}(j,:) * \mathbf{C}(k,:)) \\
\hat{\mathbf{A}}(i,:) &= \sum_{k=0}^{K} \mathbf{C}(k,:) * \sum_{j=0}^{J} \mathbf{X}_{(1)}(i,j,k)\mathbf{B}(j,:)
\end{aligned}$$

Figure 3.1: Derivation of an alternative formulation of a MTTKRP presented by Smith and Karypis [4].

against their CSF with and without their tiling scheme.

The SPLATT/CSF library presented by Smith and Karypis only implement the necessary operations to compute CPDs, but in the future could expand the library to support more tensor operations. This might be some indication of the complexity and amount of code that is required to support tensor operations for arbitrary dimensions and across all modes of operation.

Chapter 4

Implementation of Sparse Tensor Formats

In Chapter 2, we gave a general introduction to the Coordinate and Compressed Sparse Fiber formats as well as tensor-matrix products and the MTTKRP operation. In this chapter, we explore some details specific to our implementations of the tensor formats that provide context for the results in Chapter 5.

Both tensor-matrix products and the MTTKRP operation require special handling for each format and for each mode of operation. In Sections 4.1 and , we explain general features of the format implementations. To describe the algorithmic nuances, we use pseudocode in Sections 4.3 and 4.4 to explain the details of each operation at a lower level.

4.1 Coordinate Format

The Coordinate format consists of storing each non-zero value of a tensor and its i, j, k coordinates. We store each value and its in coordinates contiguously in an element data type for locality and store all these elements in an array. Additionally, we assume elements are always sorted in some mode dependent manner.

A multithreaded implementation is necessary to achieve the best performance for tensor-matrix products and the MTTKRP operation on today's multiprocessor machines. As we will describe in more detail in Section 4.3 and 4.4, it's important to allow threads to operate on independent tensor slices to avoid parallel synchronization

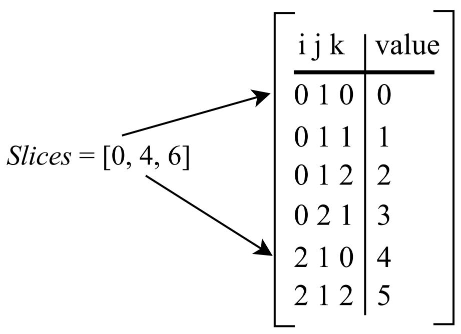


Figure 4.1: The *slices* array provides pointer to each new slice in the sorted Coordinate tensor. It allows for cheap partitioning of the tensor for multithreaded operations.

and improve temporal locality. To accomplish this for Coordinate tensors, elements are sorted in a mode dependent order and an auxiliary *slices* array is constructed. This *slices* array, introduced by this work, serves a similar purpose to the row pointer array in CSR by storing indices into the *values* array for the start of new slices. A Coordinate tensor using a *slices* array is shown in Figure 4.1. To our knowledge, it is the first work using this technique for Coordinate tensors. It requires storing an additional array with a value for each slice in the tensor. The additional storage overhead is negligible when compared to the overall footprint of tensors. The process of accessing the data of this structure is described in Algorithm 2.

Algorithm 2 Coordinate Accessing Values

```
1: function COORD-ACCESS($X$)
2:     sort $X$ by the mode of operation
3:     parallel for $s$ in 0 to $X$.sliceCount do
4:         for $i$ in $X$.slices[$s$] to $X$.slices[$s+1$] do
5:             $e \leftarrow X$.elements[$i$]
6:     end parallel for
```

In Algorithm 2, the input tensor is partitioned into slices using the *slices* array. To do this, the tensor is first sorted in a mode dependent order in line 2. Assume it is a $k - j - i$ ordering. The *slices* array then contains indices into the *values* array for each new k coordinate value. The tensor slices are partitioned to different threads in line 3 by assigning the s variable, which is used to define slice ranges. In line 4, all the i values defined on the range between s and $s + 1$ in the *slices* array belong to the same slice. This allows different threads to access independent slices of the tensor. From here, the current data value and its coordinates are stored in element e.

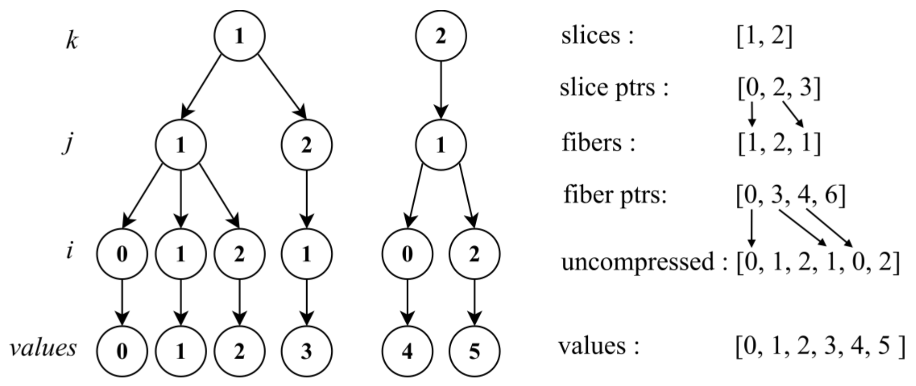


Figure 4.2: A $3 \times 3 \times 3$ sparse tensor stored in CSF2 format. The left shows the tree like representation achieved by CSF2, where the right shows how CSF2 stores the data in arrays. Arrows between arrays show where the pointers index into the next dimension of the tensor.

4.2 Compressed Sparse Tensor Format

Our implementation called Compressed Sparse Fiber Two (CSF2) is based on Smith and Karypis' CSF, which generalizes the techniques used by CSR and CSC for arbitrary tensors [6]. Although described for a third-order tensor, it is applicable to arbitrary tensors.

When compressing a tensor, first we much choose the compression order. For this work, assume the chosen compression order is $k - j - i$. This means k dimension is compressed first, then the j dimension, and the i dimension is left uncompressed for $\mathbf{X} \in \mathbb{R}^{I \times J \times K}$. The last dimension is uncompressed as an index value is stored for each non-zero value. If the last dimension was compressed, alike index values of i would be compressed when another non-zero value also shared its k and j index values. In other words, two values would share all the same coordinates. During the compression phase, multiple non-zero values with the same i, j, k coordinates should not exist, which in turn prevents any compression on the last dimension.

Our implementation of CSF, CSF2, uses six arrays and they are shown in Figure 4.2. Each compressed dimension, in this case k and j, use an array to store its coordinate values and a pointer array to store indices into the next dimension's arrays. Each new index value of the k dimension represents a new slice of the tensor. For the j dimension, each new index represents a new fiber. One of the two remaining arrays stores the coordinate values for the i dimension, which again is uncompressed. The last array is a value array storing all the non-zero values.

Algorithm 3 CSF2 Accessing Values

```
1: function CSF2-ACCESS($X$)
2:     parallel for $s$ in 0 to $X$.sliceCount do
3:         $k \leftarrow X$.slices[$s$]
4:         for $f$ in $X$.slicePtrs[$s$] to $X$.slicePtrs[$s + 1$] do
5:             $j \leftarrow X$.fibers[$f$]
6:             for $v$ in $X$.fiberPtrs[$f$] to $X$.fiberPtrs[$f + 1$] do
7:                 $i \leftarrow X$.valueCoordinates[$v$]
8:                 value $\leftarrow X$.values[$v$]
9:     end parallel for
```

Algorithm 3 shows how the CSF2 arrays can be iterated over to access the underlying values of a $k - j - i$ compressed tensor. Line 2 assigns threads independent tensor slices by assigning the s slice variable an id in the range between 0 and the number of unique slices in the tensor. Line 3 uses s to get the index value for the outermost dimension, k, from the *slices* array. The s variable is also used to index into the $slicePtrs$ array. Values stored in the $slicePtrs$ array define ranges in the $fibers$ and $fiberPtrs$ arrays. Fiber values between two consecutive $slicePtrs$ values represent separate fibers that all belong to the same slice. Line 4 uses this pattern to iterate over all the different fibers in the current slice. Similar to line 3, line 5 gets the j fiber coordinate using the f fiber value. Line 6 iterates over the last level of arrays, which store value data, using the same idiom used in line 4. The last coordinate i and tensor value are now accessible using the v value variable to index into the $valueCoordinates$ and $values$ array. At this point, all coordinates and their associated tensor value are accessible and computations can be performed. We will reuse this access pattern in each of the CSF2 pseudocodes.

4.3 Tensor-Matrix Product

As discussed in Chapter 2, tensor-matrix products are composed of a series of matrix-matrix multiplications, where one of the matrices is a fixed input matrix and the other matrix comes from the set of matrix slices that make up the input tensor. Our implementation currently only supports third-order tensors, but can be extended into higher dimensions. For this work, each matrix being multiplied by the tensor is assumed to be dense.

Tensor-matrix products often produce a sparse tensor, assuming the input tensor is sparse. However, the mode operated upon is contracted and made dense as a con-

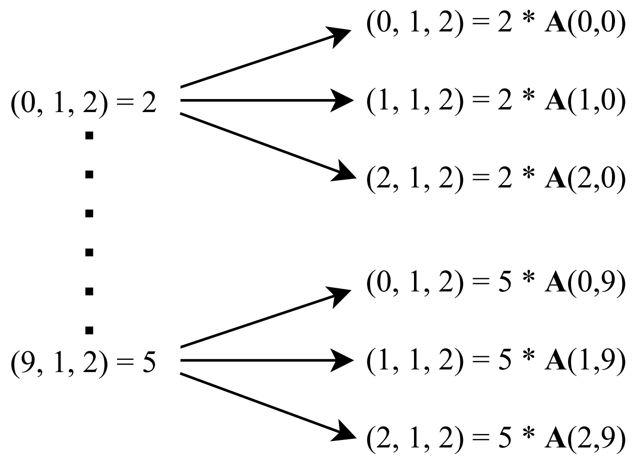


Figure 4.3: The densification and need for accumulation shown in tensor-matrix products, where the input tensor is sparse and input matrix is dense. Values with the same coordinates need to be accumulated in the final output tensor. The dense input matrix $\mathbf{A} \in \mathbb{R}^{3\times 9}$.

sequence of the dense input matrix involved in the computation. For every non-zero value, the index value corresponding to the mode of operation is replaced and made dense with index values of the rows of the input matrix. An alternative description of the densification is every non-zero value emits a dense vector, where each element is the non-zero value scaled by a value in a corresponding row of the dense matrix. Figure 4.3 helps explain this densification.

Additionally, results from different non-zero values need to be accumulated as the different results may have the same coordinates after contraction. Figure 4.3 shows the need to accumulate values. In this example of a mode-i tensor-matrix product, the first mode is being contracted. Non-zero values that share j and k index values need to be accumulated. The original index values $i = 0$ and $i = 9$ are replaced by the index values of the rows of the matrix. More generally, each fiber of the mode of operation with multiple non-zero values will need to accumulate the results in the sparse output tensor.

The sparse output tensor is likely a Coordinate format like structure because iteratively building a CSF or similar compressed structure on the fly would be too expensive. Assuming the output is a Coordinate tensor, there is still the issue of accumulating values. When we are ready to store a value and its associated coordinates, where exactly are they inserted into to output tensor? Unlike dense structures, there is no one to one mapping between a set of coordinates and an index into the output tensor value array. Three possible options to go from coordinates to a unique index into the output tensor are hash the coordinates, accumulate values over fibers, or merge results after computation.

We avoid hashing because of the hashing overhead is incurred each time the output tensor is updated. For such a frequently used operation, hashing proved to be slow. Hashing could potentially be effective in some scenarios, but was found to be inefficient when tested.

To accumulate over the fibers, the input tensor must be sorted in the correct order to allow traversing the fibers of the correct mode. For example with a mode-i tensor-matrix product, we need to traverse the mode-i column fibers, which are accessed by keeping the j and k index variables constant and iterating over all i index values. To do this efficiently, the values of the fiber need to be stored contiguously and can be achieved by sorting (or compressing) the tensor in a $k - j - i$ or $j - k - i$ fashion. All that is important is that the mode of operation is sorted last. However, we can also less efficiently accumulate over mode-i fibers if the tensor is sorted by $k - i - j$. To do so, we can accumulate values in a temporary vector indexed by the j index values. Accumulating when i is the first sorted mode is possible, but not advised in practice as it requires dense per thread matrix workspaces of size $J \times K$.

Merging values after computation is a last resort. Without accumulating over fibers or hashing, the output tensor will likely have many values with the same coordinates that need to be merged. This is an expensive process involving sorting the larger output tensor and adding values with the same coordinates. Despite being a simple process, it often takes longer than the original computation as there could be many variable number of duplicates making parallelizing the merge more complex and costly.

So far tensor-matrix products have been described in with an untransposed matrix. For our implementations and evaluation, we assume the matrix has already been transposed for improved locality.

4.3.1 Coordinate Tensor-Matrix Product

A pseudocode describing the implementation details for tensor-matrix products is included in Algorithm 4. It describes a Coordinate tensor-matrix products for all modes. In line 3, the tensor is resorted based on the mode of operation, which includes recalculating the *slices* array. Once resorted, non-zero values for the fibers of the mode of

Algorithm 4 Coordinate Tensor-Matrix Products All Modes

```
1: function COORD-TMP($X$, $A$)
2:     Output: $Y$
3:     sort $X$ by the mode of operation
4:     parallel for $s$ in 0 to $X$.sliceCount do
5:         for $i$ in $X$.slices[$s$] to $X$.slices[$s+1$] do
6:             $e \leftarrow X$.elements[$i$]
7:             $m_{ind} \leftarrow$ index for mode of operation of $e$
8:             if $e$ is in new fiber then
9:                 for $r$ in $A$.cols do
10:                    set coordinates of $Y(e.i, e.j, e.k)$ with $r$
11:                    initialize output pointer $p$ for $Y(e.i, e.j, e.k)$
12:            for $r$ in $A$.cols do
13:                $p[r]$ += $e$.value * $A(m_{ind}, r)$
14:    end parallel for
15:    return Y
```

operation are stored contiguously. Lines 4 functions to iterate over the slices in parallel, whereas line 5 provides an actual index into the elements array. Using this index, we get the element we will be accessing in line 6 and also get the specific index value for the mode of operation in line 7. The index value for the mode of operation is needed to access the value in the input matrix that will be used to scale the tensor value. Line 8 checks to see if the current element is in a new fiber. If the element is in a new fiber, then the coordinates in the output tensor need to be set for this new fiber. This is implicitly done in line 10, which also sets the coordinate for the mode of operation to r. Line 11 sets up a pointer to the correct output position in the output tensor. The pointer into the output tensor allows other elements in the same fiber of later iterations to accumulate their results in the same region. After the initialization of the output region is finished for each new fiber, the actual computation is performed in lines 12 and 13, using the value of the current element and each value in the corresponding column on the input tensor. Results are stored in the output tensor using the output pointer.

The implementation for Coordinate tensor-matrix products relies on resorting the Coordinate tensor. The overhead of resorting might be too costly for some small tensors, but it worked well for our use cases as we focused on large sparse tensors. By resorting, non-zero values for the fibers of the mode of operation are stored contiguously. This results in a temporary free accumulation and a simple algorithm. Output Coordinate tensors are preallocated and can store the maximum number of output non-zero values, which is *nnz* in the input tensor times the number of rows in the matrix. This allows for direct indexing by multiple threads. As a result, the output tensor has many zero values that need to be removed after computation.

4.3.2 CSF2 Tensor-Matrix Product

CSF2 tensor-matrix products are performed with a single fixed compressed tensor. And as with Coordinate tensor-matrix products, the output Coordinate tensors are preallocated to allow for easier mutlithreaded updates. Assume the CSF2 input tensor is compressed in $k - j - i$ order. Algorithms 5, 6 and 7 provide pseudocode for CSF2 tensor-matrix products for modes i, j, and k respectively.

Algorithm 5 CSF2 Tensor-Matrix Products Mode-i

```
 1: function CSF2-TMP($X$, $A$)
 2:     Output: $Y$
 3:     parallel for $s$ in 0 to $X$.sliceCount do
 4:         $k \leftarrow X$.slices[$s$]
 5:         for $f$ in $X$.slicePtrs[$s$] to $X$.slicePtrs[$s+1$] do
 6:             $j \leftarrow X$.fibers[$f$]
 7:             for $r$ in 0 to $A$.cols do
 8:                 $accum \leftarrow 0$
 9:                 for $v$ in $X$.fiberPtrs[$f$] to $X$.fiberPtrs[$f+1$] do
10:                     $i \leftarrow X$.valueCoordinates[$v$]
11:                     $accum \mathrel{+}= X$.values[$v$] $* A(i, r)$
12:                 set $Y(f, r)$ coordinates with $(r, j, k)$
13:                 $Y(f, r)$.value $\leftarrow accum$
14:     end parallel for
15:     remove any zeros from Y
16:     return $Y$
```

Algorithm 5 shows the most performant mode of a tensor-matrix product for this specific compression of CSF2. Line 3 partitions the slice iteration space to threads by assigning thread-local s variables. As seen in line 4, the s value is used to get the k coordinate value from the *slices* array. Line 5 iterates over the fibers in the current slice and assigns the current fiber to f. Line 7 iterates over the column values of the input matrix, which allows accumulation of results for the output tensor. Line 8 initializes an accumulation variable. Line 9 iterates over the values of the current fiber and stores the current value index to v. Line 11 adds to the accumulation variable the current value times the value in the input tensor for the current column and row. The row comes from the mode of operation, which in this case is i and assigned in line 10. Line 12 and 13 set up the coordinates for the output position and store the result. The output position used in lines 12 and 13 is calculated as $f * A.\text{cols} + r$. Line 15 removes any zero from Y so storage and any further calculations with the tensor are efficient.

Algorithm 6 CSF2 Tensor-Matrix Products Mode-j

```
 1: function CSF2-TMP($X$, $A$)
 2:     Output: $Y$
 3:     parallel for $s$ in $X$.sliceCount do
 4:         $k \leftarrow X$.slices[$s$]
 5:         for $r$ in 0 to $A$.cols do
 6:             $accum \leftarrow$ 0-vector of size mode-$i$
 7:             for $f$ in $X$.slicePtrs[$s$] to $X$.slicesPtrs[$s+1$] do
 8:                 $j \leftarrow X$.fibers[$f$]
 9:                 for $v$ in $X$.fiberPtrs[$f$] to $X$.fiberPtrs[$f+1$] do
10:                     $i \leftarrow X$.valueCoordinates[$v$]
11:                     $accum[X.\text{values}[v]] += X.\text{values}[v] * A(j,r)$
12:             for $f$ in $X$.slicePtrs[$s$] to $X$.slicesPtrs[$s+1$] do
13:                 $j \leftarrow X$.fibers[$f$]
14:                 for $v$ in $X$.fiberPtrs[$f$] to $X$.fiberPtrs[$f+1$] do
15:                     $i \leftarrow X$.valueCoordinates[$v$]
16:                     set $Y(f,v)$ coordinates with $(i,r,k)$
17:                     $Y(f,v)$.value $\leftarrow accum[X.\text{values}[v]]$
18:     end parallel for
19:     remove any zeros from Y
20:     return $Y$
```

Algorithm 6 for mode-j starts similarly to mode-i by partitioning the slice iteration space to threads in line 3 and getting the k coordinate value in line 4. However, mode-j must accumulate over all the fibers in a slice, the iteration over the input matrix's columns is moved to the second loop as shown in line 5. In this columns iteration loop, there are two separate iterations over the fibers in the current slice with one on line 7 and the other on line 12. In the first iteration over the fibers, the partial results involving each tensor value and input matrix value are accumulated in the accumulation vector in line 11. The accumulation vector is indexed using the i coordinate value. The second iteration over the fibers serves to output the accumulated values into the output tensor in lines 16 and 17. The output tensor is indexed using the current fiber value, f, and r, the current column of the input matrix involved in each calculation. Finally, any zero values are removed from the output tensor in line 19.

Mode-k tensor-matrix product, shown in Algorithm 7, has the simplest access pattern. Lines 3-8 use the same access pattern for accessing the coordinates and value as

Algorithm 7 CSF2 Tensor-Matrix Products Mode-k

```
 1: function CSF2-TMP($X$, $A$)
 2:     Output: $Y$
 3:     parallel for $s$ in 0 to $X$.sliceCount do
 4:         $k \leftarrow X$.slices[$s$]
 5:         for $f$ in $X$.slicePtrs[$s$] to $X$.slicesPtrs[$s + 1$] do
 6:             $j \leftarrow X$.fibers[$f$]
 7:             for $v$ in $X$.fiberPtrs[$f$] to $X$.fiberPtrs[$f + 1$] do
 8:                 $i \leftarrow X$.valueCoordinates[$v$]
 9:                 for $r$ in 0 to $A$.cols do
10:                     set $Y(v, r)$ coordinates with $(i, j, r)$
11:                     $Y(v, r)$.value $\leftarrow X$.values[$v$] $* A(k, r)$
12:     end parallel for
13:     remove any zeros from Y
14:     merge redundant coordinate values of $Y$
15:     return $Y$
```

explained for Algorithm 3. Because there is no accumulation over columns for this mode, line 9 iterates over the columns and is the innermost loop. Line 10 set the coordinates for the current value and line 11 stores the computed value. The computed values being stored in the output tensor are only partial factors for the final results. Therefore, tensor values with redundant coordinate values are merged in line 14 after removing any zero tensor values in line 13.

With this fixed compression order, the CSF2 implementation accumulates over fibers when performing tensor-matrix products for mode-i and mode-j. Mode-i is the most performant mode because it can accumulate over a fiber with contiguous values. Mode-j requires iterating over all the fibers in a slice twice and accumulating in a larger workspace. However when performing over mode-k, to avoid constructing an even larger per thread $J \times K$ workspace, CSF2 emits results to different locations in the output tensor for every value. This results in redundant coordinates throughout the output tensor that require a costly merging phase.

4.4 MTTKRP

The MTTKRP operation produces a dense matrix from one input tensor and two input matrices. Using the MTTKRP algorithm in Figure 3.1, entire rows of the output matrix are computed at a time. As MTTKRP is a mode dependent operation, each mode is computed slightly differently for a fixed structure.

To examine some of the more low level details of MTTKRP, consider a mode-i MTTKRP, $\mathbf{A} = \mathbf{X}_i(\mathbf{C} \odot \mathbf{B})$, where $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$ and $\mathbf{X} \in \mathbb{R}^{I \times J \times K}$. Regardless of the sorting/compression, the index into the output matrix for MTTKRP is always computable for a specific non-zero value as the index value for the mode of operation of the current non-zero value and the column being computed. For example, with a mode-1 MTTKRP,

$$\mathbf{A}(i,:) = \mathbf{C}(k,:)\mathbf{X}(i,j,k)\mathbf{B}(j,:)$$

, where the constraints i, j, and k are imposed by $\mathbf{X}(i,j,k)$.

For optimal execution, all non-zero values that share the same i should be stored contiguously to improve locality. Additionally, when parallelized this allows partitioning different slices, each with a unique i index value, to separate threads. By doing this, the MTTKRP operation can be performed without locks.

The MTTKRP operation is often cheaper to compute for the same input tensor than performing a tensor-matrix product. This is largely because MTTKRP results are stored into a dense output matrix. This smaller structure has better cache performance and index values do not require explicit storage as they are implied by the value's position in the matrix.

4.4.1 Coordinate Tensor MTTKRP

With the Coordinate format, the overhead associated with resorting the tensor for each mode is too expensive for a MTTKRP operation. To avoid this cost, the Coordinate tensor is kept fixed. Assume the fixed $k - j - i$ ordering is kept for all modes of operation as we did for CSF2 when evaluating tensor-matrix products. When parallelizing MTTKRP for modes i and j of operation, the values are not stored in the optimal ordering for accumulation and it is more performant to scatter values into the output matrix. As different threads could be racing on the same output location in the matrix, locks must be used to ensure correct results leading to lowered performance. For mode-k, output values can be accumulated per row and safely written to the output matrix lock-free. Pseudocode for the first two modes of operation are extremely similar and are included in Algorithms 8 and 9. Pseudocode for faster lock-free mode of operation is include in Algorithm 10.

Algorithm 8 Coordinate MTTKRP Mode-i

```
1: function COORD-MTTKRP($X$, $C$, $B$)
2:     Output: $A$
3:     parallel for $s$ in 0 to $X$.sliceCount do
4:         for $i$ in $X$.slices[$s$] to $X$.slices[$s+1$] do
5:             $e \leftarrow X$.elements[$i$]
6:             LOCK
7:             for $r$ in $C$.cols do
8:                 $A(e.i, r)$ += $e$.value * $C(e.k, r)$ * $B(e.j, r)$
9:             UNLOCK
10:    end parallel for
11:    return A
```

Algorithm 8 provides pseudocode for a mode-i Coordinate MTTKRP. Line 3 partitions the tensor for parallel access to independent tensors. Line 4 iterates over all the index values in the current slice by assigning i the current value index. After getting the current element data in line 5, line 6 locks the current row of the output matrix so that other threads cannot interfere. Columns are iterated over in line 7, which allows line 8 to perform the actual computation using the current column, the current value and its coordinates, and the input matrices. Values are stored directly into the dense output matrix here as well using the i coordinate of the current value and the current column. After storing values for the current row, the row is unlocked and the algorithm continues until termination.

Algorithm 9 Coordinate MTTKRP Mode-j

```
1: function COORD-MTTKRP($X$, $C$, $A$)
2:     Output: $B$
3:     parallel for $s$ in $X$.sliceCount do
4:         for $i$ in $X$.slices[$s$] to $X$.slices[$s+1$] do
5:             $e \leftarrow X$.elements[$i$]
6:             LOCK
7:             for $r$ in $C$.cols do
8:                 $B(e.j, r)$ += $e$.value * $C(e.k, r)$ * $A(e.i, r)$
9:             UNLOCK
10:    end parallel for
11:    return B
```

For mode-j, the structure of Algorithm 9 is almost identical to that of Algorithm 8. However, line 8 changes to compute the result value based on different input matrices and the result is now stored using the j coordinate value.

There is a potential for partial aggregation for mode-j, which would reduce the amount of locking. However, for our locking scheme, which used a large pool of locks, we found that the more fine grain locking with minimal algorithmic bookkeeping was slightly more performant for the Coordinate scheme. This could be an anomaly arising from the small number of datasets tested. Future libraries will additionally need to consider the most generalizable approach. We chose to present this algorithm for its simplicity and slight performance boost for the most fair comparison.

Algorithm 10 Coordinate MTTKRP Mode-k

```
1: function COORD-MTTKRP($X$, $B$, $A$)
2:     Output: $C$
3:     parallel for $s$ in $X$.sliceCount do
4:         $accum \leftarrow$ 0-vector of size mode-$r$
5:         for $i$ in $X$.slices[$s$] to $X$.slices[$s+1$] do
6:             $e \leftarrow X$.elements[$i$]
7:             if $e$ is in new fiber then
8:                 for $r$ in $B$.cols do
9:                     $C(e.k, r)$ += $accum[r]$ * $B(j_{last}, r)$
10:                    fill $accum$ with 0's
11:            for $r$ in $B$.cols do
12:                $accum[r]$ += $e$.value * $A(e.i, r)$
13:        // Outputs any additional accumulated values
14:        for $r$ in $B$.cols do
15:            $C(e.k, r)$ += $accum[r]$ * $B(e.j, r)$
16:    end parallel for
17:    return C
```

Coordinate MTTKRP mode-k is the optimal mode for accumulating results and avoids using locks. Algorithm 10 presents pseudocode depicting the algorithm for mode-k. Line 3 partitions the tensor for parallel access to independent tensors. The accumulation vector is initialized in line 4. Line 5 serves to iterate over the values in the current slice of the tensor. Data for the current element is gathered in line 6. Line 7 checks to see if the current element is in a new fiber. If the element is in a new fiber, then the values for the current row in the accumulation vector need to be scaled by the B input matrix before they are dumped into the output matrix. Pay close attention that the B input matrix is index using the j_{last}, which is the j value associated with the last fiber. The accumulation vector is then cleared in line 10 for later use. After outputting the accumulated values for the old fiber, partial results for the new fiber are calculated and stored in the accumulation vector in line 12. The accumulation vector is indexed using the current column. Finally, after iterating over all the values in the current slice, any additional partial results stored in the accumulation vector are scaled by their corresponding values in input matrix B and stored in the output matrix.

4.4.2 CSF2 MTTKRP

As with tensor-matrix products, the CSF2 format is kept fixed when computing a series of MTTKRP operations for all modes. Assume the compression order is fixed at $k - j - i$ ordering. Parallelization of MTTKRP for CSF2 is very similar to the strategy used for tensor-matrix products. However, two modes, in this case j and k, use aggregation while mode-i scatters values.

Algorithm 11 CSF2 MTTKRP Mode-i

```
1: function CSF2-MTTKRP($X$, $C$, $B$)
2:     Output: $A$
3:     parallel for $s$ in 0 to $X$.sliceCount do
4:         $k \leftarrow X$.slices[$s$]
5:         for $f$ in $X$.slicePtrs[$s$] to $X$.slicesPtrs[$s+1$] do
6:             $j \leftarrow X$.fibers[$f$]
7:             for $v$ in $X$.fiberPtrs[$f$] to $X$.fiberPtrs[$f+1$] do
8:                 $i \leftarrow X$.valueCoordinates[$v$]
9:                 LOCK
10:                for $r$ in 0 to $C$.cols do
11:                    $A(i,r)$ += $X$.values[$v$] * $C(k,r)$ * $B(j,r)$
12:                UNLOCK
13:    end parallel for
14:    return $A$
```

As stated, mode-i scatters results for each value into the output matrix, which requires the use fine grain locking as shown in Algorithm 11. Lines 3-8 use the same access pattern for accessing the coordinates and value as explained in Algorithm 3. Because various threads could be attempting to output values to the same row in the output matrix, the output row, i, is locked in line 9 to prevent any data races. The computation for the current value and column is performed and accumulated in the output row in line 11. After all computation for the current value is completed, the row is unlocked and the algorithm continues until completion.

Algorithm 12 for mode-j aggregates partial results to reduce locking. Lines 3-9 provide the normal structure for accessing coordinate and value data from the CSF2 structure. Partial results are calculated in line 11 for the current value and all columns of the input matrix, A. After accumulating all these partial results for all the values in

Algorithm 12 CSF2 MTTKRP Mode-j

```
1:  function CSF2-MTTKRP($X$, $C$, $A$)
2:      Output: $B$
3:      parallel for $s$ in 0 to $X$.sliceCount do
4:          $k \leftarrow X$.slices[$s$]
5:          for $f$ in $X$.slicePtrs[$s$] to $X$.slicesPtrs[$s + 1$] do
6:              $accum \leftarrow$ 0-vector of size $C$.cols
7:              $j \leftarrow X$.fibers[$f$]
8:              for $v$ in $X$.fiberPtrs[$f$] to $X$.fiberPtrs[$f + 1$] do
9:                  $i \leftarrow X$.valueCoordinates[$v$]
10:                 for $r$ in 0 to $C$.cols do
11:                     $accum[r]$ += $X$.values[$v$] * $A(i, r)$
12:             LOCK
13:             for $r$ in 0 to $C$.cols do
14:                 $B(j, r)$ += $C(k, r)$ * $accum[r]$
15:             UNLOCK
16:     end parallel for
17:     return $B$
```

the current fiber, the accumulated values are scaled by their corresponding C value and stored into the output matrix. Because other threads could also be trying to output results with the same j coordinate, the writing to the output matrix must use locks on the output row.

Unlike the mode-j implementation for Coordinate structure, which uses no aggregation, the CSF2 format uses partial aggregation. The partial results are scattered into the output matrix using a lock for thread-safe updates. This proved faster for CSF2 than more frequent scattering, but still was slowed by the use of locks.

Lastly, CSF2 MTTKRP operated on mode-k required no locks as it was able to aggregate and emit complete results for rows of the output matrix at a time. This is process is shown in Algorithm 13. Line 3 partitions the tensor so threads get independent slices. Line 4 get the current k coordinate associated with the current slice and line 5 represents iterating over the fibers in the current slice s. To perform the proper aggregation, partial results must be accumulated for each fiber. The vector to store these accumulated values is initialized in line 6. Line 8 iterates over the value indices for the current fiber, f. With all of the necessary coordinates available, lines 10 and 11 perform part of the actual MTTKRP computation by multiplying the tensor value with the matrix value. This partial result is stored in the accumulation vector. The computation is completed in line 13, where it is scaled by the other input matrix value

Algorithm 13 CSF2 MTTKRP Mode-k

```
function CSF2-MTTKRP($X$, $B$, $A$)
    Output: $C$
    parallel for $s$ in 0 to $X$.sliceCount do
        $k \leftarrow X$.slices[$s$]
        for $f$ in $X$.slicePtrs[$s$] to $X$.slicesPtrs[$s+1$] do
            $accum \leftarrow$ 0-vector of size $C$.cols
            $j \leftarrow X$.fibers[$f$]
            for $v$ in $X$.fiberPtrs[$f$] to $X$.fiberPtrs[$f+1$] do
                $i \leftarrow X$.valueCoordinates[$v$]
                for $r$ in 0 to $C$.cols do
                    $accum$[$r$] += $X$.values[$v$] * $A(i, r)$
            for $r$ in 0 to $C$.cols do
                $C(k, r)$ += $B(k, r)$ * $accum$[$r$]
    end parallel for
    return $C$
```

and stored into the output matrix.

Chapter 5

Performance Results

5.1 Experimental Methodology

To evaluate the different formats we compare the performance of Coordinate and CSF2 based tensors for both tensor-matrix products and the MTTKRP operation. We additionally compare the MTTKRP performance of these formats against of Smith and Karypis' CSF format on the same datasets. Their library altered by increasing the number of locks, which resulted in better performance. Additionally, slight modifications that didn't alter CSF's performance were made in order to collect the proper timing data.

The Coordinate and CSF2 data structures and operation implementations were written in C++ using the shared memory parallelism features of OpenMP. All index values were 64-bit integers and data values were double-precision floats. GCC 4.8.4 was used to compile the code and tuned with -mtune=corei7-avx and level 3 optimizations. Tensor-matrix products and MTTKRP operations were evaluated on an Intel Xeon CPU E5-2695 v2 at 2.40GHz with 32 KB L1-D, 32 KB L1-I, 2048 KB L2 and 30,720 KB L3 caches. For each experiment the tensor operations were parallelized using 16 threads each with their own core and all on one chip. For tensor-matrix operations, matrices consisted of 8 rows ($R = 8$) and were transposed prior to timing. When evaluating MTTKRP, matrices consisted of 16 columns ($R = 16$).

The datasets used for evaluation consisted of the original Netflix Prize dataset and synthetic datasets constructed from the Netflix Prize dataset [11]. To give a more accurate comparison to CSF, which removes empty slices from tensors as they are being compressed, the original Netflix dataset was preprocessed to remove all empty slices.

Dataset	**I**	**J**	**K**	**NNZ**	**Sparsity**
Netflix	480K	18K	2K	100M	5.79e−6
Double I	960K	18K	2K	200M	5.79e−6
Double J	480K	36K	2K	200M	5.79e−6
Double K	480K	18K	4K	200M	5.79e−6
Double All	960K	36K	4K	200M	1.46e−6

Table 5.1: K represents thousand, M stands for million, NNZ represents the number of non-zero values and Sparsity is the ratio of non-zero values to total possible values.

This results in each dimension of the tensor becoming dense. That is there is at least one non-zero value for each row, column and tube fiber that makes up the tensor. Despite its dense dimensions, the tensor is still sparse because a majority of values in the tensor as a whole are zero. Three synthetic datasets were constructed by doubling the Netflix Prize dataset in one dimension for each dimension and then an additional dataset constructed by doubling all dimensions. The synthetic datasets contain double the number of non-zero values as the original dataset. Table 5.1 shows the dimensional sizing of all the datasets tested.

The sparsity of the original Netflix Prize dataset varied throughout the tensor. For instance, movies on average had over 5,000 ratings, but some movies had as few as three ratings. There was even an instance of one user submitting over 17,000 reviews whereas the average user submitted around 200 reviews. The sparsity factor of the original Netflix, Double I, Double J, and Double K datasets was 5.79e−6, whereas the sparsity factor of the Double All data set was 1.45e−6.

5.2 Storage Comparison

Figure 5.1 shows the storage requirements for the datasets in the different sparse tensor formats. CSF2 gets 1.88× compression ratio when compared to the Coordinate equivalent on the original Netflix Prize dataset. Smith and Karypis's reported a similar compression ratio of 1.88× for CSF in their paper [6]. Similar compression ratios were seen for all datasets. All tested tensors are larger than the last level of cache as they each measure several gigabytes.

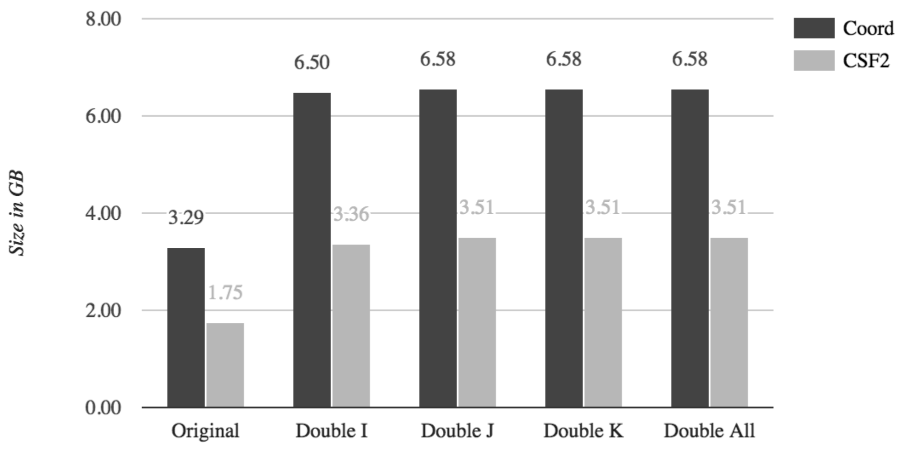
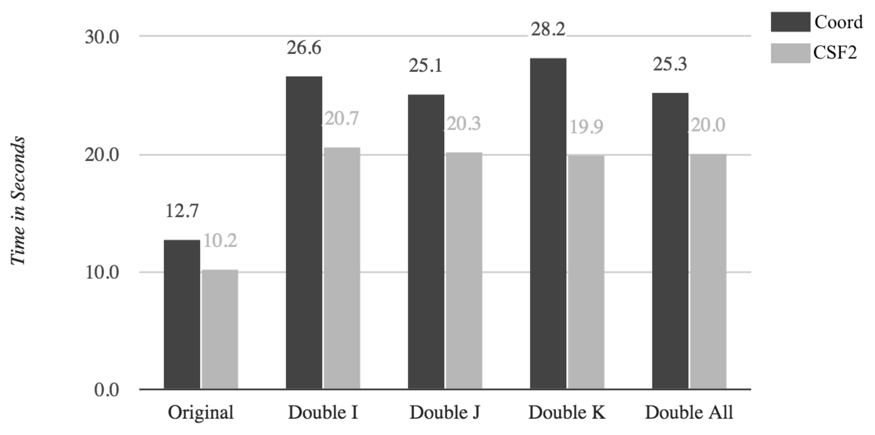


Figure 5.1: Storage requirements for the Coordinate and CSF2 formats measured in gigabytes.

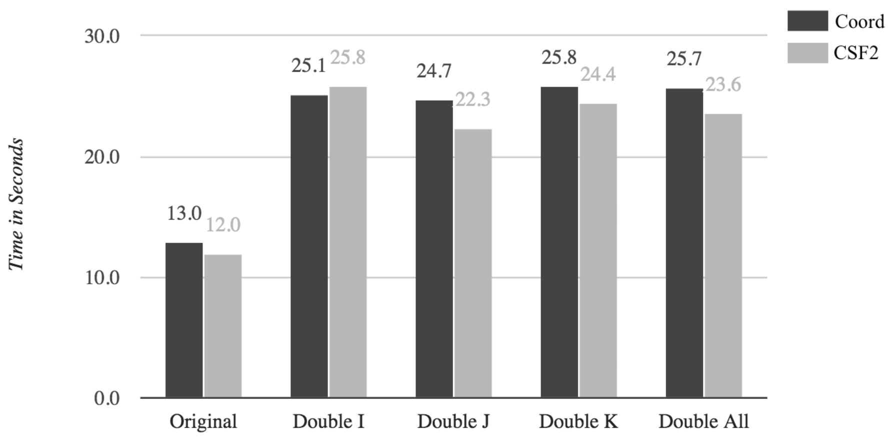
5.3 Tensor-Matrix Product Results

We evaluated tensor-matrix products for Coordinate and CSF2 formats. CSF was not tested as tensor-matrix products of this form were not implemented in the CSF library. Figure 5.2 shows the timing results for tensor-matrix products on specific modes. Figure 5.3 shows the timing results accumulated over all modes. The time for any necessary resorting, merging or removal of zeros is counted in the timing.

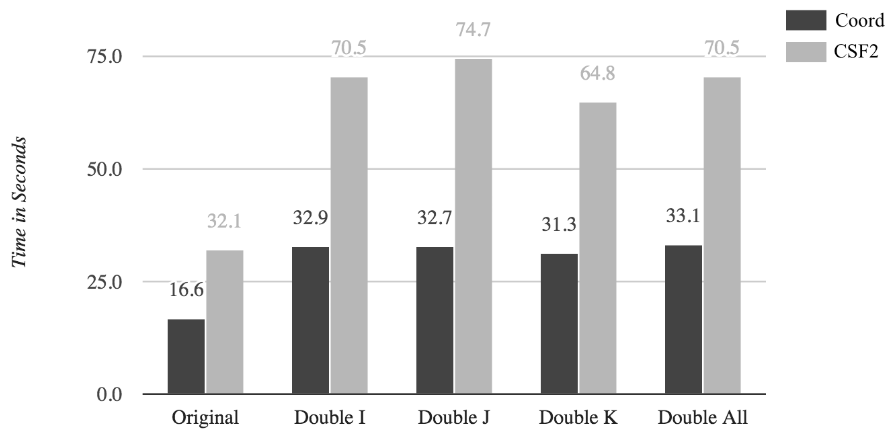
The compression of CSF2 provides a slight performance boost for the first two modes of tensor-matrix products, but the merging required for mode-k proved to be a large performance hit. Because of the merge requirement, CSF2 proved to be much slower than the Coordinate format for mode-k products. In fact, Figure 5.3 shows the merge procedure was enough to make the aggregate runtime across all three modes longer for CSF2 than Coordinate. For reference, resorting the Coordinate structure took anywhere from 10 to 15% of its runtime. When CSF2 merges in mode-k, the merge takes roughly 75% of the runtime for that mode. This provides evidence for the benefits of the simplicity of a Coordinate based tensor. While although it may not perform the most efficiently for every case, the Coordinate's flexibility proves its usefulness when evaluated under many conditions.



(a) Mode-i Tensor-Matrix Timing Results



(b) Mode-j Tensor-Matrix Timing Results



(c) Mode-k Tensor-Matrix Timing Results

Figure 5.2: Tensor-Matrix Product timing results by mode.

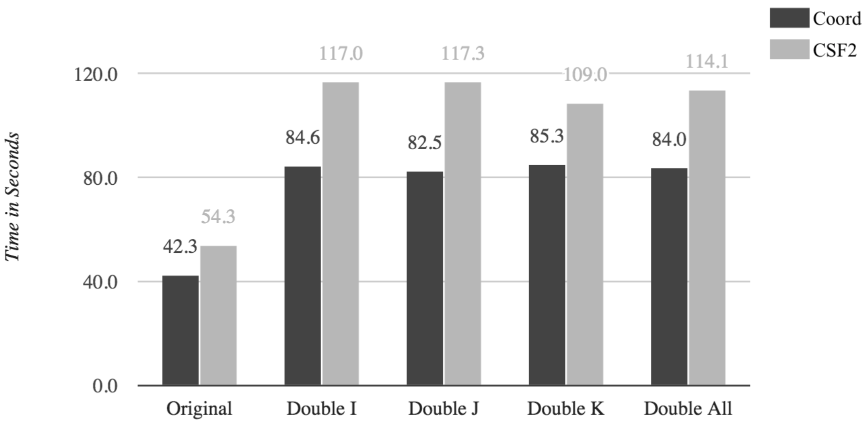


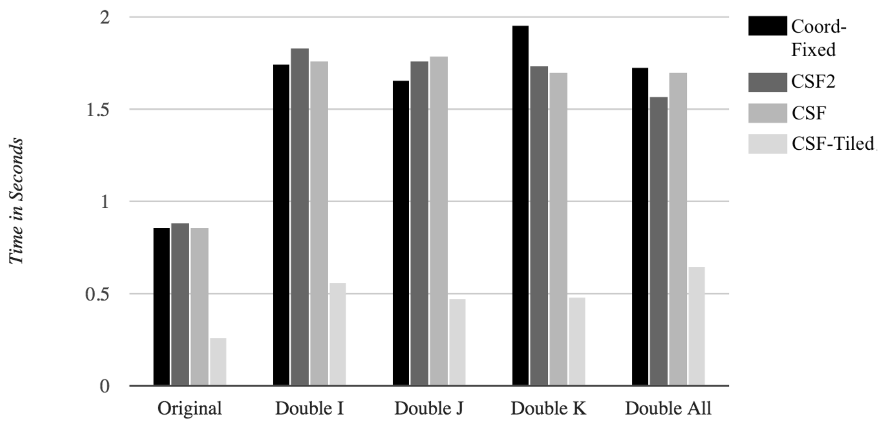
Figure 5.3: Aggregate Tensor-Matrix Product timing results for all three modes.

5.4 MTTKRP Results

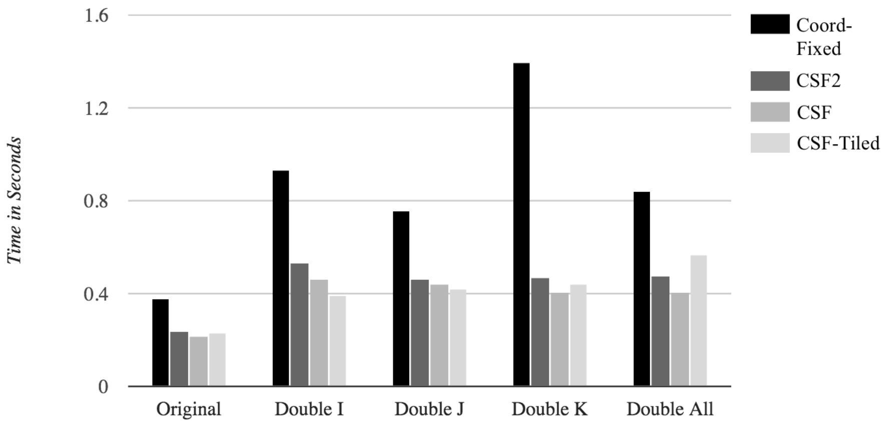
In addition to the Coordinate and CSF2 formats, we evaluated MTTKRP for Smith and Karypis's CSF with and without tiling. Tiling requires some additional overhead to store the smaller tiles, but improves cache locality during computation. Due to the complexity of the tiling scheme, it was not implemented for Coordinate or CSF2. Future tensor libraries should spend effort researching such matters as it proved effective for CSF. Smith and Karypis' paper should be referenced for further details[6]. Figure 5.4 displays the timing results for specific modes while Figure 5.5 contains an aggregate view of the MTTKRP timing results across all modes.

For this evaluation, all formats used a fixed sorting/compression. Unlike for tensor-matrix products, the Coordinate sorting was kept fixed due to its relatively large overhead cost for the lighter MTTKRP operation. Evaluating the performance of executing a series of MTTKRP operations across all modes is a realistic scenario as it is performed every iteration of a CPD. Therefore, a format which is performant across all modes of operation is critical.

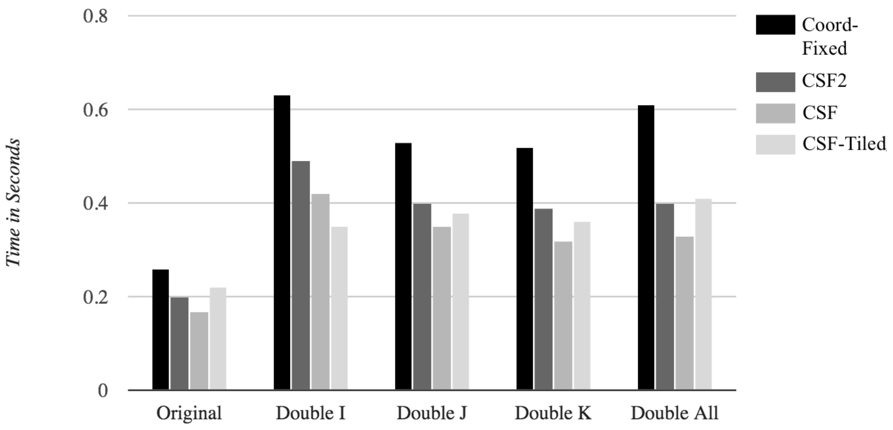
CSF-tiled proved to be by far the most performant scheme as a result of its better cache locality. Coordinate with a fixed sort, CSF2 and CSF without tiling perform similarly for mode-i, but the Coordinate format falls behind for the other modes of operation. CSF2 and CSF perform fairly similar for all evaluations, which is expected as they are intended to implement a similar data structure. Any differences are likely the result of specific implementation details.



(a) Mode-i MTTKRP Results



(b) Mode-j MTTKRP Results



(c) Mode-k Tensor-Matrix Timing Results

Figure 5.4: MTTKRP timing results by mode.

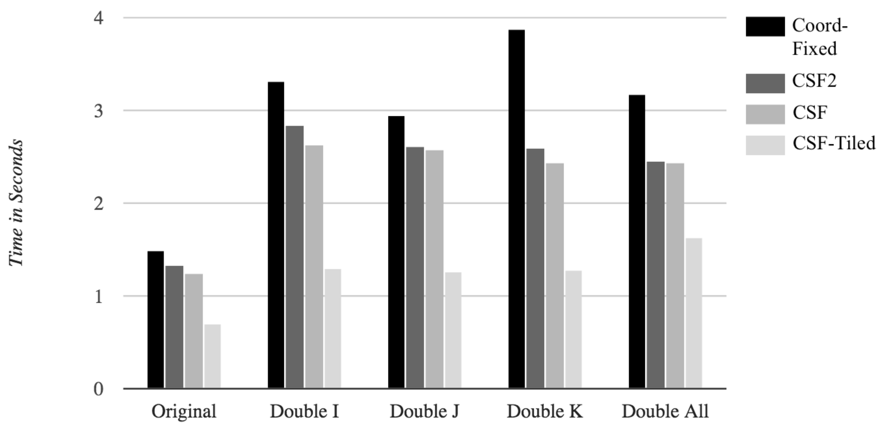


Figure 5.5: Aggregate MTTKRP timing results for all three modes.

The lack of performance of the Coordinate format for MTTKRP is a result of it being fixed to a particular sorting like the other formats, but while also operating on an uncompressed tensor. The additional storage reduces the amount of relevant data in cache resulting additional capacity misses throughout computation. In the worst case for a single mode, Coordinate performed on mode-j ran 2.98x slower than the next format. In the worst case for all modes, Coordinate ran 1.49x slower than the next slowest format. As the MTTKRP operation is fundamental in the CPD, the Coordinate's poor performance supports the need for a compressed tensor format that performs well across all modes of operation.

Chapter 6

Discussion

One of the major motivations for this work was see how a multithreaded Coordinate scheme compares to the multithreaded CSF implementation for the MTTKRP operation as well as tensor-matrix operations. Smith and Karypis provide a unfair comparison between a single threaded Coordinate scheme and a multithreaded CSF for the MTTKRP operation [6]. By evaluating our multithreaded Coordinate implementation against their CSF implementation, we get a less biased view of these two types of formats. Additionally by implementing CSF2, which is based on CSF and achieves similar performance on the MTTKRP operation, it is reasonable to compare our multithreaded Coordinate implementation indirectly to CSF by comparing it to CSF2 for tensor-matrix products. Again, this is necessary because CSF only supports those operations directly used in tensor decompositions and does not support tensor-matrix products. Using these results, as they are disclosed in Chapter 5, there is no definitive answer as to which format should be used as the standard for tensor libraries, but rather they show the potential benefits of a hybrid approach. In this chapter, we provide some guidance on how these formats can be used efficiently.

The Coordinate and CSF2 formats were considered for their use on sparse tensors and should not be used for dense tensors. Dense tensors would see no benefit from such storage arrangements as they unnecessarily store coordinates for each value.

The real debate between the Coordinate based format and the CSF format only arises for non-uniformly distributed tensors, where there is a large potential for compression. The Coordinate format should be used for all uniformly distributed sparse tensors because uniformly distributed tensors don't offer much opportunity for compression. Coordinate tensors lower the fixed storage overhead, which results in a smaller storage footprint. Additionally, the Coordinate format is more flexible, which allows it be quickly resorted for optimal mode-dependent operations.

Which format, Coordinate or CSF, should be used for a non-uniform sparse tensor? It depends on the operations being performed on the tensor, but both will likely need to be supported by a high performant tensor library.

Operations that produce sparse tensors should store the results using a Coordinate structure because compressing an output tensor on the fly would be too expensive. This makes the Coordinate structure fundamental to tensor libraries.

The main concern of the input tensor format for operations that produce sparse tensors is the ability to aggregate values. The later reading or operating on an output tensor with scattered values would be prohibitively expensive. Therefore, an aggregation must be performed either during or after computation. Aggregating after computation is much more expensive and results in poor performance. The determining factor for whether this aggregation may take place during computation is the sort/compression order of the tensor and the mode of operation. If the operation is only performed on one mode and the initial tensor compression order agrees with this mode of operation, using a compressed tensor will almost certainly be more efficient than using the Coordinate format. The storage savings of compression reduces cache capacity misses allowing for faster execution. But for cases where these operations are performed across multiple modes, the Coordinate structure can be quickly resorted, which allows these mode-dependent operations to always be performed on the optimal ordering. We saw this to be faster in aggregate as we compared Coordinate with CSF2's tensor-matrix results in Figure 5.2.

However, there are several potential ways of alleviating the shortcomings of the compressed tensor in this use case. A straightforward one would be implementing a faster merge routine. The merge used for the described evaluations was a relatively simple single threaded approach. The lack of uniformity in the number of values that needs to be merged makes multithreaded implementations complex. Additionally, designing the compressed tensor data structure with this merging requirement in mind could allow for more optimal performance. Another approach could be eliminating the need for a merge altogether. Similar to how Coordinate is easily resorted, finding a way to cheaply recompress a CSF like data structure could provide the best of both worlds for these operations. The CSF structure could be recompressed using a more ideal compression ordering for a given mode of operation and thereby alleviating the need for merging all together. A breakthrough in any of these approaches might lead to a more universal use of a compressed format.

When considering operations that emit dense tensors, most often in the form of matrices and vectors, aggregating values can always be done on the fly as every set of coordinates maps to one defined and easily indexed location in the output tensor. This makes these operations as a whole less expensive than those operations that emit sparse tensors because the output tensors are often smaller and completely fit into cache. Because of these reasons, the ease of resorting the Coordinate structure is less valuable. Additionally, resorting the Coordinate tensor has shown to be too expensive for these relatively cheap operations. Because of this, CSF/CSF2 will likely perform better than the Coordinate structure for operations that emit dense, usually small and low dimensional tensors.

Lastly, it should not be ignored that the fastest times by a wide margin were from Smith and Karypis' CSF tiled implementation [6]. Tiling was shown to greatly improve the performance of compression based tensors, but would also be applicable to a Coordinate tensor. The Coordinate tensor could benefit more from the tiling scheme as they generally suffer from a lower cache hit rate. This gives tiling a larger margin for improvement. We did not implement tiling for the Coordinate or CSF2 formats due to its complexity. Because of its performance benefits, a mature tensor library should make tiling a high priority despite the additional complexity it adds. Smith and Karypis discuss their tiling techniques used to improve cache efficiency when they introduced SPLATT [4]. Despite already having positive results, there is likely additional research in this area that could prove extremely useful.

Chapter 7

Conclusion and Future Work

In this work, we have shown the tradeoffs of using different sparse tensor formats for different tensor operations. The motivation for this work was to provide future tensor libraries with the necessary background information to determining their sparse tensor storage scheme.

These formats have only been evaluated in three dimensions by two operations. Although these formats were chosen for their scalability and generalizability into higher dimensions, the lack of a variety of operations being tested in an arbitrary dimensional setting is a weak point. This lack of testing largely stems from the amount of code required to implement these operations across all modes and for arbitrary dimensions. As shown by this work, these operations change algorithmically when operating on different modes. This blowup in code is only compounded when there is a need to support multiple formats, which this work has shown to be useful. Given this issue in the amount of code needed to support multiple formats for a suite of tensor operations across all modes, it is almost certain that a high performant tensor library will need to employ compiler and code generation techniques to have any chance of persevering maintainability and performance.

The work required to build a highly performant sparse tensor library is daunting as many deserving projects are. Yet such a system will be built because there is an increasing need. An optimized and fully functional tensor library can have a large impact on the future of data science. The power and beauty of tensors come from its simplicity in storage and in computation. It is only fair that engineering a similarly powerful system to make use of its beauty should require some dedication and persistence.

Bibliography

[1] D. M. Dunlavy, T. G. Kolda, and W. P. Kegelmeyer, "Multilinear algebra for analyzing data with multiple linkages," Tech. Rep. SAND2006-2079, Sandia National Laboratories, April 2006.

[2] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," Tech. Rep. SAND2006-7592, Sandia National Laboratories, December 2006.

[3] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, pp. 455–500, September 2009.

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

[5] F. L. Hitchcock, "The expression of a tensor or a polyadic as a sum of products," *J. Math. Phys*, vol. 6, no. 1, pp. 164–189, 1927.

[6] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, (New York, NY, USA), pp. 5:1–5:7, ACM, 2015.

[7] F. G. Gustavson, "Some basic techniques for solving sparse systems of linear equations," in *Sparse matrices and their applications*, pp. 41–52, Springer, 1972.

[8] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," *SIAM Journal on Scientific Computing*, vol. 30, pp. 205–231, December 2007.

[9] J. R. Gilbert, C. Moler, and R. Schreiber, “Sparse matrices in matlab: Design and implementation,” *SIAM J. Matrix Anal. Appl.*, vol. 13, pp. 333–356, Jan. 1992.

[10] C.-Y. Lin, Y.-C. Chung, and J.-S. Liu, “Efficient data compression methods for multidimensional sparse array operations based on the ekmr scheme.,” *IEEE Trans. Computers*, vol. 52, no. 12, pp. 1640–1646, 2003.

[11] J. Bennett and S. Lanning, “The netflix prizes.” Categories and Subject Descriptors I.2.6 [Machine Learning]: Engineering applications -applications of techniques. General Terms Experimentation, Algorithms.