ADVANCES IN TENSOR DECOMPOSITIONS: FAST MATRIX MULTIPLICATION ALGORITHMS AND PARALLEL ADAPTIVE COMPRESSION TECHNIQUES

BY

JOÃO VICTOR DE OLIVEIRA PINHEIRO

A Thesis Submitted to the Graduate Faculty of

WAKE FOREST UNIVERSITY GRADUATE SCHOOL OF ARTS AND SCIENCES

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Computer Science

May 2025

Winston-Salem, North Carolina

Approved By:

Grey Ballard, Ph.D., Advisor

Aditya Devarakonda, Ph.D., Chair

Frank Moore, Ph.D.

Ramakrishnan Kannan, Ph.D.

ACKNOWLEDGEMENTS

Antes de mais nada gostaria de dedicar este trabalho aos meus pais, que sob muito sol, fizeram-me chegar até aqui, na sombra. Tudo que eu tenho eu devo a eles, em compensação, tudo o que eu faço, faço pensando neles. Carlo Giovanni e Doris Pinheiro, amo vocês mais do que minha propria vida. Desejo a minha irmã, Ana Carolina Pinheiro, o mundo inteiro, pois ela merece isso tudo e um pouco mais.

I would like to thank the Wake Forest Computer Science Department for funding my graduate work. This work was funded by the NSF Grant No. CCF-1942892 and OAC-210692. I am truly appreciative of the Sparsitute for fostering a wonderful environment of talented and creative people.

I am thankful to Dr. Paul Pauca for being my first ever computer science professor as he was fundamental for my upbringing in this field. Even after finishing his assignments quite early on he would always encourage my intellectual curiosity. I would like to thank Dr. Samuel Cho for pushing me outside my comfort zone multiple times and teaching many important lessons in the process. I would like to thank Dr. Errin Fulp for always being a comforting person to talk to in the good and bad times. His door was always open, and a friendly conversation always awaited on the other side. I would like to thank Dr. Pete Santago for his steady guidance realism when I needed it most, and bringing humor to our everyday conversations. I would like to thank Dr. Daniel Cañas, whose tough words in our early interactions challenged me more than he may have realized. Today, I am grateful not only for the motivation he sparked, but also for the friendship and respect we now share. I would like to thank Dr. Natalia Khuri for her no-nonsense wisdom and the countless practical lessons she shared with me. I would like to give special thanks to Cody Stevens, whose impact on this department—and on me personally—is immeasurable. Our countless conversations, both technical and personal, fueled my passion and kept me going when I needed it most.

Prior to my move to the Department of Computer Science I was an undergraduate student in the Department of Mathematics. I would now like to thank the following people from that Department. I would like to express my tremendous gratitude to Dr. Lynne Yengulalp for giving my first ever job in the Math and Stats Center. This tutor position she has given me has ignited my passion for teaching and helping others. I would like to thank Dr. Stephen Robinson for his kindness and friendliness outside of class, and it was during his course that I realized that my true passion lied in the Mathematics. I would like to thank Dr. Pratyush Mishra for being an exceptional research collaborator. Though we often approached the same problems from the distinct lenses of mathematics and computer science, it was through navigating that difference that I learned the true art of communicating ideas across disciplines. I would like to thank Dr Leandro Lichtenfelz and Dr. Leonardo Cella as they have shown me the difference it makes seeing someone of equal ethnical background as a rolemodel. I would like to thank Dr. John Gemmer, my undergraduate advisor, whose honest and thoughtful guidance played a key role in both my decision to pursue a master's in computer science and my Ph.D. applications this cycle. He always knew exactly what to say—never sugarcoating, never discouraging—just clear, grounded advice that helped me find the path where I truly belong.

Most importantly, I would like to thank my committee for supporting me in every step of the way. I would like to thank Dr. Ramakrishnan Kannan, whose kindness and humility left a lasting impression on me during my first academic conference. At a time when I felt out of place and overwhelmed, his welcoming presence reminded me that academia is not only about research, but also about community. I would like to thank Dr. Frank Moore for not only always laughing the hardest at my jokes, and for his transparency and help in crucial times. I thought I knew how to code until I begain working with Dr. Aditya Devarakonda. Through our work together, he taught me what it means to write code at a research level with great rigor and purpose. Last, but not least, is Dr. Grey Ballard. I have a feeling that in my entire

life, I will meet few people like him. Just witnessing his genuine passion for what he does—teaching, researching, and mentoring—is simply indescribably inspiring. I am truly grateful and honored to have been mentored by him.

I couldn't possibly have made it without all the friends that were part of my life, especially for the past two years. I would like to thank all my friends from both cohorts I was a part of. Especially Cade Wiley, Nikhil Rajkumar, Ziyue (Parry) Yang, William Bailey, Alejandro Gonzalez Rubio, e meu irmão Raniery Mendes. Additionally, I am truly grateful to have met the amazing Whitener family, particularly Nathan Whitener who has always been by my side since day one of this program. I also dedicate this work to my amazing partner, Andie Barnes, one of the most hardworking, intelligent, and kindhearted people I have ever had the pleasure of knowing.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF ABREVIATIONS

CP Decompositions

CP: Canonical Polyadic

Tucker Decompositions

DT: Dimention Trees

EVD: Eigenvalue Decomposition

HOOI: Higher Order Orthogonal Iterations

HOSI: Higher Order Subspace Iterations

HOSVD: Higher Order Singular Value Decomposition

LLSV: Left Leading Singular Values

ST-HOSVD: Sequentially Truncated Higher Order Singular Value Decomposition

SVD: Singular Value Decomposition

ABSTRACT

Tensors are essential in modern-day computational and data sciences. This work presents recent advances in tensor decompositions, which are techniques that break down complex high-dimensional arrays into smaller structured components. There are two projects presented in this thesis, each with its own abstract and chapter.

Searching For Cyclic Invariant Fast Matrix Multiply Algorithms using the CP Decomposition: Fast matrix multiplication algorithms correspond to exact CP decompositions of tensors that encode matrix multiplication of fixed dimensions. This 3-way matrix multiplication tensor has cyclic symmetry: the entry values are invariant under cyclic permutation of the indices. The CP decomposition of Strassen's original fast matrix multiplication algorithm for 2x2 matrices is cyclic invariant, which means a cyclic permutation of the CP factors results in the same CP components, just in a different order. We describe how to search for cyclic invariant solutions using the damped Gauss-Newton optimization method along with heuristic rounding techniques. We not only summarize the algorithms discovered so far but also attempt to search for further symmetries in these algorithms by describing the requirements for an algorithms to admit such symmetries.

Parallel Rank-Adaptive Higher-Order Subspace Iteration for the Tucker Decomposition: Higher Order Orthogonal Iteration (HOOI) is an iterative algorithm that computes a Tucker decomposition of fixed ranks of an input tensor. In this work we modify HOOI to determine ranks adaptively subject to a fixed approximation error, apply optimizations to reduce the cost of each HOOI iteration,

and parallelize the method in order to scale to large dense datasets. We show that HOOI is competitive with the Sequentially Truncated Higher Order Singular Value Decomposition (STHOSVD) algorithm, particularly in cases of high compression ratios. Our proposed rank-adaptive HOOI can achieve comparable approximation error to STHOSVD in less time, sometimes achieving a better compression ratio. We demonstrate that our parallelization scales well over thousands of cores and show using three scientific simulation datasets that HOOI outperforms STHOSVD in high-compression regimes. For example, for a 3D fluid-flow simulation dataset, HOOI computed a Tucker decomposition 82x faster and achieved a compression ratio 50% better than STHOSVD's.

Introduction

1.1 Tensors and Their Subparts

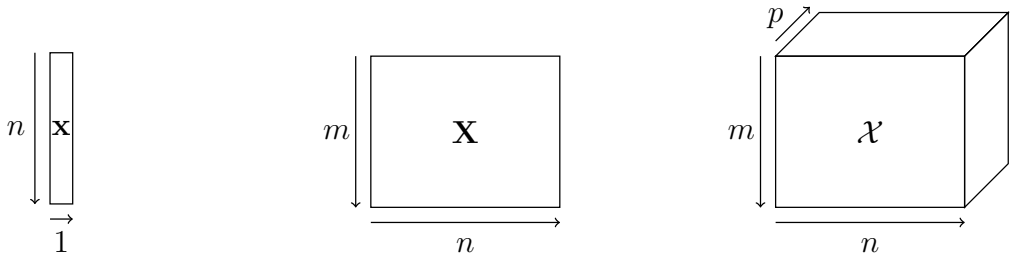
1.1.1 What Is A Tensor?

There is no widely agreed upon definition of a tensor; different fields define it differently. For the purposes of this work, we define tensors through the definition widely used in the field of computer science. A **tensor** is a d-way array, where d is referred to as the order of the tensor. We use the following notational conventions. The set of real values is denoted as $\mathbb{R}$. Letters m, n, p, q, r are used to represent sizes (or simply $n_1, \cdots, n_d$) and letters i, j, k, ℓ are used to represent indices (or simply $i_1, \cdots, i_d$). For the sake of simplification, let $[n] \equiv [1, \cdots, n]$, and furthermore let $[m] \otimes [n] \equiv \{(i, j) \mid i \in [m],\, j \in [n]\}$. Some lower order tensors have other names:

- A **scalar** is a 'zero-dimensional' tensor. This is any number $x \in \mathbb{R}$.
- A **vector** is a one-dimensional array of scalars. This is visualized in figure 1.1a.

We represent vectors by lowercase boldface roman letters. If $\mathbf{x}$ is a real-valued vector of size n, then we write that $\mathbf{x} \in \mathbb{R}^n$. Entry $i \in [n]$ of $\mathbf{x}$ is denoted as $\mathbf{x}(i)$, or compactly as $\mathbf{x}_i$. A vector is a tensor of order 1. Instead of referring to them as 1-way tensors, they will simply be referred to as vectors.

- A **matrix** is a two-dimensional array of scalars, such as a collection of vectors. This is visualized in figure 1.1b. We represent matrices by uppercase boldface roman letters. If $\mathbf{X}$ is a real-valued matrix of size $m \times n$, then we write $\mathbf{X} \in \mathbb{R}^{m \times n}$. The matrix entry $\mathbf{X}(i, j)$ represents the i^{th} entry of column vector j. More generally, entry $(i, j) \in [m] \otimes [n]$ of $\mathbf{X}$ is denoted as $X(i, j)$ or compactly as $x_{i,j}$. A matrix is a tensor of order 2. Instead of referring to them as order 2 tensors, they will simply be referred to as matrices.



(a) Vector $\mathbf{x} \in \mathbb{R}^n$ is a 1-way tensor

(b) Matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ is a 2-way tensor

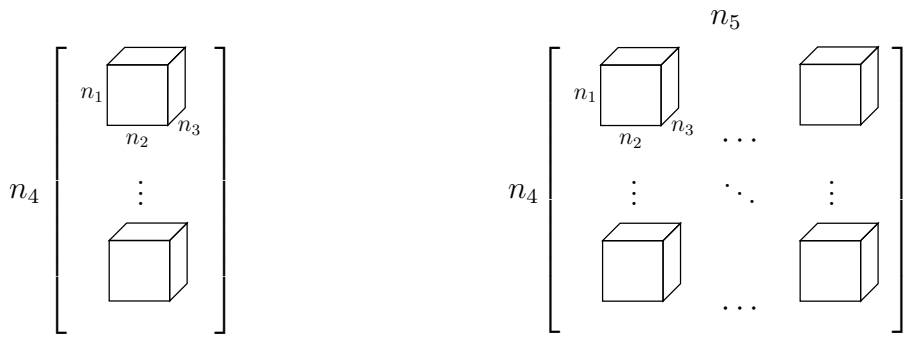
(c) Tensor $\mathcal{X} \in \mathbb{R}^{m \times n \times p}$ is a 3-way tensor

Figure 1.1: Tensors of orders one, two, and three

If we have a three-dimensional array of scalars, then we have a higher-order tensor. Tensors of order 3 or greater are denoted by uppercase mathematical calligraphy

letters: $\mathcal{X}$. This is visualized in figure 1.1c. If $\mathcal{X}$ is a real-valued tensor of size $m \times n \times p$, then we write $\mathcal{X} \in \mathbb{R}^{m \times n \times p}$. For instance, given a set of m objects, each of which has n features, measured under p different scenarios, the tensor entry $\mathcal{X}(i, j, k)$ represents the j^{th} feature of object i measured in scenario k. More generally, entry $(i, j, k) \in [m] \otimes [n] \otimes [p]$ of $\mathcal{X}$ is denoted as $\mathcal{X}(i, j, k)$ or compactly as $x_{i,j,k}$. We refer to each dimension as a **mode**. Of a 3-way tensor, we say that mode 1 is of size m, mode 2 of size n, and mode 3 of size p. If all modes have the same size, we call this tensor **cubical** or refer to it as a tensor with uniform dimensions.

As mentioned earlier, any tensor of order greater than or equal to three is simply referred to as a higher-order tensor. But we begin to run out of letters to describe its size and index its modes. This is when we resort to subscripts mentioned earlier when notation was discussed. Figure 1.2 illustrates 4-way and 5-way tensors. There, we can visualize the recursive nature of tensors. A 4-way tensor is can be visualized as an array of 3-way tensors. Similarly, a 5-way tensor can be visualized as a matrix of 3-way tensors. In applications, a simulation in 3 spatial dimensions through time that tracks a certain number of variables produces a 5-way tensor of data. To consolidate some of this information, we refer to table 1.1.



(a) A 4D Tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$

(b) A 5D Tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4 \times n_5}$

Figure 1.2: Tensors of orders four and five

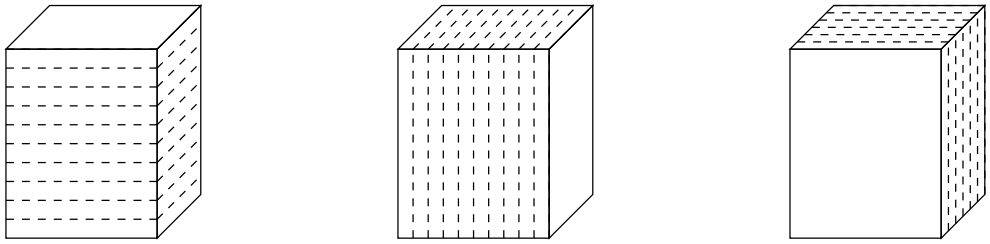
Description	**Size**	**Order**	**Notation**	**Entry**
Scalar	1	0	x	x
Vector	n	1	$\mathbf{x}$	$x(i)$ or x_i
Matrix	$m \times n$	2	$\mathbf{X}$	$X(i,j)$ or x_{ij}
3-way tensor	$m \times n \times p$	3	$\mathcal{X}$	$X(i,j,k)$ or x_{ijk}
4-way tensor	$n_1 \times n_2 \times n_3 \times n_4$	4	$\mathcal{X}$	$X(i_1,i_2,i_3,i_4)$ or $x_{i_1 i_2 i_3 i_4}$
d-way tensor	$n_1 \times n_2 \times \cdots \times n_d$	d	$\mathcal{X}$	$X(i_1,i_2,\ldots,i_d)$ or $x_{i_1 i_2 \cdots i_d}$

Table 1.1: Tensor Notation by Order

1.1.2 Slices and Fibers

A slice of a 3-way tensor $\mathcal{X} \in \mathbb{R}^{m \times n \times p}$ is a 2-way subtensor (which is a matrix). The i^{th} **horizontal slice** is a matrix of size $n \times p$ given by $\mathcal{X}(i,:,:)$. The j^{th} **lateral slice** is a matrix of size $m \times n$ given by $\mathcal{X}(:,j,:)$. The k^{th} **frontal slice** is a matrix of size $m \times n$ given by $\mathcal{X}(:,:,k)$. The three types of slices for 3-way tensors are shown in figure 1.3.

The concept of slices is generalizable for higher-order tensors of order greater than



(a) Horizontal Slices $\mathcal{X}(i,:,:)$ (b) Lateral Slices $\mathcal{X}(:,j,:)$ (c) Frontal Slices $\mathcal{X}(:,:,k)$

Figure 1.3: Two-way slices of a 3-way tensor

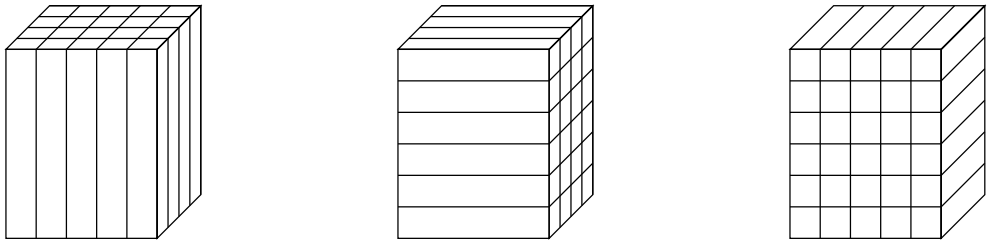
or equal to 4, those are called hyperslices. Tensor fibers are the analogs of rows and columns of matrices. The main difference between matrx rows and columns and tensor fibers is that tensor fibers are always oriented as column vectors when used in calculations. For a 3-way tensor, of size $m \times n \times p$, we have the following:

- The **mode-1 fibers** of length m, also known as **column fibers**, range over all indices in the first mode, holding the second and third indices fixed. In other words, there are np column fibers of the form $\mathbf{x}_{:jk} \in \mathbb{R}^m$. This can be visualized on Figure 1.4a.

- The **mode-2 fibers** of length m, also known as **row fibers**, range over all indices in the second mode, holding the first and third indices fixed. In other words, there are mp row fibers of the form $\mathbf{x}_{i:k} \in \mathbb{R}^n$.This can be visualized on Figure 1.4b.

- The **mode-3 fibers** of length m, also known as **tube fibers**, range over all indices in the third mode, holding the first and second indices fixed. In other

words, there are mn tube fibers of the form $\mathbf{x}_{ij:} \in \mathbb{R}^p$. This can be visualized on figure 1.4c.



(a) Column Fibers $\mathbf{x}_{:jk}$ (b) Row Fibers $\mathbf{x}_{i:k}$ (c) Tube Fibers $\mathbf{x}_{ij:}$

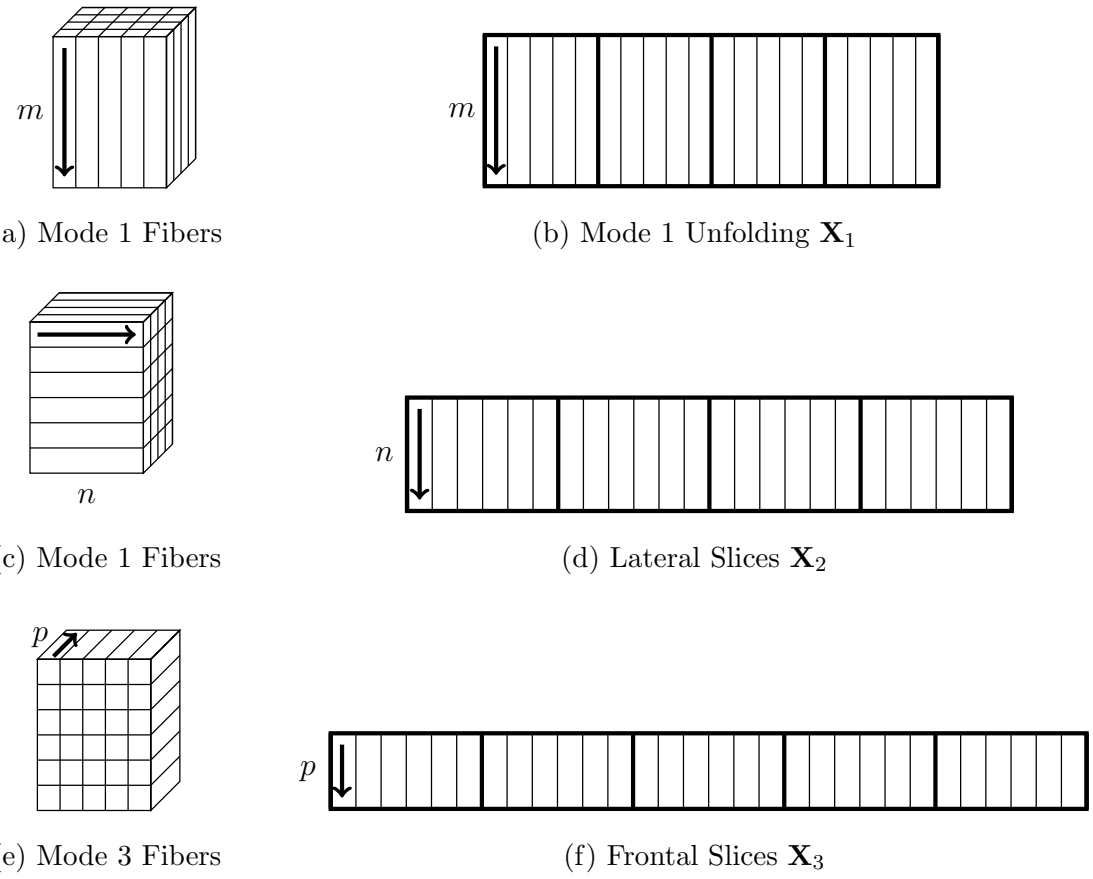
Figure 1.4: Fibers of a 3-way tensor

1.1.3 Tensor Mode-k Unfoldings

The elements of a tensor can be rearranged to form various matrices in a procedure referred to as **unfolding**, also known as **matricization** since the result is always a matrix. A particular unfolding of interest is the mode-k unfolding which is defined as a matrix whose columns are the mode-k fibers of that tensor. The notation for a mode-k unfolding of a tensor $\mathcal{X}$ is $\mathbf{X}_{(k)}$. Figure 1.5 illustrates the mode-k unfoldings of a 3-way tensor.

1.1.4 Types of Tensor Multiplication

There exists a variety of tensor operations. Given that multiplication involves two entities, this subsection aims to systematically explore the different forms of tensor multiplication by incrementally increasing the dimensionality of the operands. For



(a) Mode 1 Fibers (b) Mode 1 Unfolding $\mathbf{X}_1$

(c) Mode 1 Fibers (d) Lateral Slices $\mathbf{X}_2$

(e) Mode 3 Fibers (f) Frontal Slices $\mathbf{X}_3$

Figure 1.5: Unfoldings of a 3-way tensor

instance, the multiplication of two order-zero tensors is trivial, as it corresponds to the multiplication of two scalars. By increasing the dimensionality on one side of the multiplication while keeping the other fixed, we arrive at the multiplication of an order-zero tensor with an order-one tensor—that is, a scalar and a vector. In this case, the operation simply scales the entries of the vector by the scalar. Proceeding further, we consider the multiplication of two order-one tensors, i.e., vector-vector products. In this context, two primary products are defined: the inner product and the outer product, both of which are described below. As we explore these tensor

multiplication types, two key concepts emerge: the matching of dimensions and the contraction of those matched dimensions.

Vector Inner Products

The inner product of two same-sized vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ produces a scalar, is denoted as $\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^\intercal \mathbf{b}$, and is defined as

$$\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^\intercal \mathbf{b} = \begin{bmatrix} a_1 & \cdots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \sum_{i=1}^{n} a_i b_i. \tag{1.1}$$

The vector 2-norm is defined as the square root of the inner product of a vector with itself: $\sqrt{\langle \mathbf{a}, \mathbf{a} \rangle}$. The computational complexity of vector inner products is $O(n)$.

Vector Outer Products

In contrast to vector inner products, which are a reductive operation generating a scalar, vector outer products are an expansive operation that generates a matrix. Vector outer products are defined for multiple vectors, which is why we avoid the common notaion of $\mathbf{a}\mathbf{b}^\intercal$. We start by defining the outer products of two vectors. Given two vectors $\mathbf{a} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^m$, their vector outer product is defined as

$$\mathbf{C} = \mathbf{a} \circ \mathbf{b} \in \mathbb{R}^{m \times n}, \text{ where } c_{ij} = a_i b_j, \forall (i,j) \in [m] \otimes [n], \text{ or}$$

$$\begin{bmatrix} a_1 b_1 & \cdots & a_1 b_n \\ \vdots & \ddots & \vdots \\ a_m b_1 & \cdots & a_m b_n \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_m \end{bmatrix} \begin{bmatrix} b_1 & \cdots & b_n \end{bmatrix}. \tag{1.2}$$

Therefore, the outer product of two vectors generates a 2-way tensor. In general, the outer product of d vectors generates a d-way tensor, and it is written as $\mathcal{X} = \mathbf{x}_1 \circ \cdots \circ \mathbf{x}_d$. The computational complexity of the outer product of two vectors of size m and n respectively is $O(mn)$. In general, the computational complexity of d vectors of respective size of $n_1, \cdots, n_d$ is $O(\prod_{i=1}^{d} n_d)$.

Matrix-Vector Products

Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and vector $\mathbf{x} \in \mathbf{R}^n$, the matrix-vector product is defined as

$$\mathbf{y} = \mathbf{A}\mathbf{x} \in \mathbb{R}^m, \text{ where } y_i = \sum_{j=1}^{n} a_{ij} x_j \text{ for all } i \in [m], \text{ or}$$

$$\begin{bmatrix} a_{11}x_1 + \cdots + a_{1n}x_n \\ \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}. \tag{1.3}$$

The computational complexity of the matrix-vector product is $O(mn)$.

Matrix-Matrix Products

Given two matrices $\mathbf{A} \in \mathbb{R}^{m \times p}$ and $\mathbf{B} \in \mathbb{R}^{p \times n}$, the matrix-matrix product is defined as

$$\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times n}, \text{ where } c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}, \; \forall (i,j) \in [m] \otimes [n]. \tag{1.4}$$

The computational complexity of the matrix-matrix product is $O(mnp)$. In general, the computational complexity of multiplying two same-sized matrices of size n by n is $O(n^3)$. A simple and yet noteworthy example is 2 by 2 matrix multiplication:

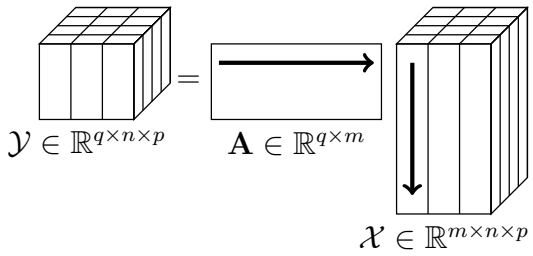
$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}, \tag{1.5}$$

which shows that matrix multiplication is performed by inner products of all pairs of the rows of $\mathbf{A}$ with the columns of $\mathbf{B}$.

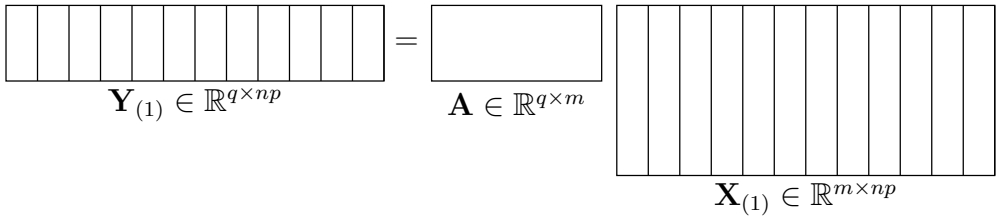
Tensor-Times-Matrix (TTM) Products

The tensor-times-matrix (TTM) product is a mode-wise multiplication denoted as $\mathcal{Y} = \mathcal{X} \times_k \mathbf{A}$ where $\mathcal{X}$ is a tensor, k is the mode for the TTM, and $\mathbf{A}$ is a matrix. This product can be transformed into a matrix-matrix product using tensor unfoldings, as we can define the product as $\mathbf{Y}_{(k)} = \mathbf{A}\mathbf{X}_{(k)}$. As the columns of a mode-k unfolding are the fibers of mode k, we can also interpret the TTM product in terms of the matrix acting on the fibers. In other words, the TTM multiplies each mode-k fiber of $\mathcal{X}$ by $\mathbf{A}$. In figure 1.6b we see the notion that a TTM multiplies each column fiber of $\mathcal{X}$ by

the rows of A, and in figure 1.6b we see the notion that this is equivalent to unfolding the input and output tensors and performing a regular matrix multiplication.



(a) Tensor form: the first row of $\mathbf{A}$ and first mode-1 fiber of $\mathcal{X}$ are emphasized with arrows.



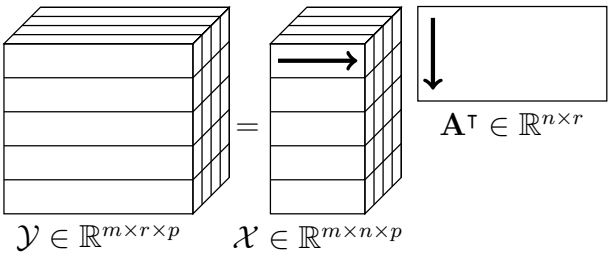
(b) Matrix form.

Figure 1.6: Mode-1 TTM (along column fibers)

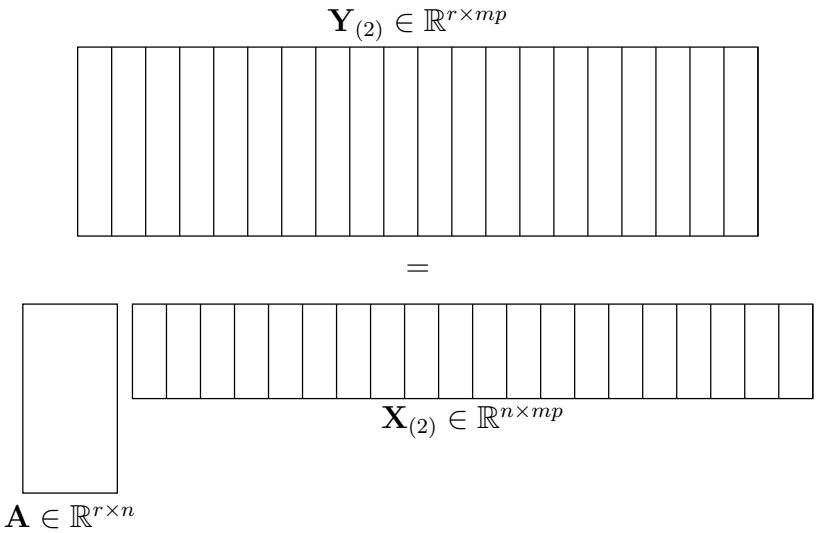
Mathematically, the Tensor Times Matrix (TTM) operation has the same structure os matrix multiplication. Mode-1 TTMs, in particular, align closely with the intuitive understanding presented in figure 1.6. However, practical implementations become more complex for higher modes, as the tensor's memory layout significantly influences the computation. The technical details involved in these practical considerations are beyond the scope of this work. For a more in-depth treatment of TTM implementation, see [1].

Because of this omission, figure 1.7 showcases the mathematical way of representing a TTM on mode 2, which is not how it is performed computationally. Similarly to

mode-1 TTM, we visualize it in its tensor format in figure 1.7a as applying the xmatrix $\mathbf{A}$ to the row fibers of $\mathcal{X}$. If we wish to visualize this TTM as matrix multiplication, there is no transpose, and figure 1.7b has the same format as figure 1.6b.



(a) Tensor form: the first row of $\mathbf{A}$ and first mode-2 fiber of $\mathcal{X}$ are emphasized with arrows.



(b) Matrix Form

Figure 1.7: Mode-2 TTM (along row fibers)

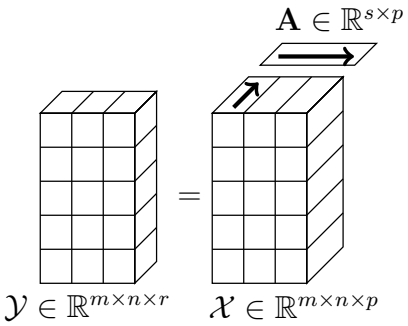
It is in the matrix-multiplication format that one can understand best the nature of the details of how TTM operations are performed computationally. The best mode to understand the complications that arise in practical TTM operations is the last mode of a tensor. For the 3-way case, that can be seen in figure 1.8. Notice that the

matrix multiplication format of this TTM in figure 1.8b has been adapted to have the transpose of all elements involved. This is to avoid any data memory movement which is deemed expensive. Again, more details of this can be found on [1]. For the 3-way case, the cost of a TTM of a cubical tensor of size n and a matrix of size n by n is $O(n^4)$. For the general case, $\mathbf{A} \in \mathbb{R}^{r \times n_k}$ and a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_k \times \cdots \times n_d}$, the computational complexity is $O(r \cdot \prod_{i=1}^{d} n_d)$.

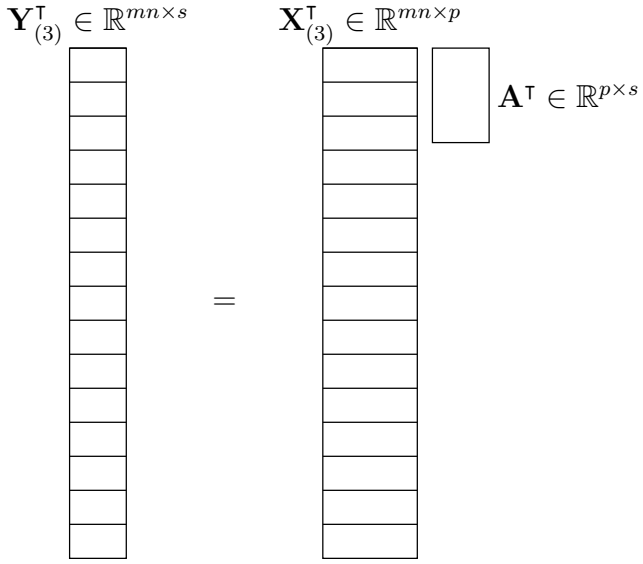
Tensor-Times-Tensor (TTT) products

The last type of Tensor multiplication required for this work is the Tensor-Times-Tensor (TTT) multiplication, which is also known as Tensor Contraction. Before proceeding to this next case—which is considerably more challenging to visualize using the illustrative methods employed thus far—it is worthwhile to briefly revisit the types of multiplication discussed up to this point.

Notice how from the first type of multiplication covered, we required the inner dimensions to match. For vector-vector products, there were two types of multiplication as there are two ways we can match the dimensions. An array $\mathbf{a} \in \mathbb{R}^n$ can be multiplied by an array $\mathbf{b}$ of the same length either through $\langle \mathbf{a}, \mathbf{b} \rangle$ (inner product) or $\mathbf{a} \circ \mathbf{b}$ (outer product). In other words, we could either perform a 1 by n times n by 1 inner product or an n by 1 times a 1 by n outer product. The former *contracts* the inner dimension of size n to produce a 1 by 1 scalar, and the latter *contracts* the inner



(a) Tensor form: the first row of $\mathbf{A}$ and first mode-3 fiber of $\mathcal{X}$ are emphasized with arrows.



(b) Matrix Form

Figure 1.8: Mode-3 TTM (along tube fibers)

dimension of size 1 to produce an n by n matrix. The same idea of contracting the matching inner dimensions can be seen in matrix-vector products where we match the second dimension of matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ to the first dimension of vector $\mathbf{x} \in \mathbb{R}^n$, where they are then contracted to produce a vector of size m. In matrix-multiplication we match the inner dimenions of two matrices, contract those two matching dimensions, and the size of the output matrix is the outer dimensions of the two matrices. In a mode-k TTM we match the columns of matrix to the mode-k fibers of a tensor $\mathcal{X}$, then the k^{th} mode is contracted. The idea of the generalized tensor contraction is the same; we contract some subset of the matching modes of two tensors.

Consider two tensors, $\mathcal{X} \in \mathbb{R}^{m \times n \times p}$ and $\mathcal{Y} \in \mathbb{R}^{p \times q \times r}$. The last mode of $\mathcal{X}$ matches the size of the first mode of $\mathcal{Y}$, so we can contract along those modes. The result is a tensor $\mathcal{Z} \in \mathbb{R}^{m \times n \times q \times r}$ defined by

$$\mathcal{Z}(i_1, i_2, j_1, j_2) = \sum_{k=1}^{p} \mathcal{X}(i_1, i_2, k) \cdot \mathcal{Y}(k, j_1, j_2), \; \forall (i_1, i_2, j_1, j_2) \in [m] \otimes [n] \otimes [q] \otimes [p] \quad (1.6)$$

As mentioned earlier, the nature of the drawings presented so far are not useful for visualizing this form of multiplication. Tensor contractions can get complicated not only because of this change in visualization, but also in the notation for d-way tensor contractions. Since the details go beyond the scope of this work, they will be omitted. For this work, it suffices to know that as long as one or more modes of two tensors match, a contraction is possible along those modes.

Other Types of Matrix-Matrix Products

We now go over a few other matrix products that come up in the context of tensor decompositions.

Matrix Hadamard Products

Given two same-sized matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$, their Hadamard product, also known as element-wise product, is

$$\mathbf{C} = \mathbf{A} * \mathbf{B} \in \mathbb{R}^{m \times n}, \text{ where } c_{ij} = a_{ij} b_{ij}, \forall (i,j) \in [m] \times [n], \text{ or}$$

$$\begin{bmatrix} a_{11}b_{11} & \cdots & a_{1n}b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & \cdots & a_{mn}b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{bmatrix}. \tag{1.7}$$

Matrix Kronecker Products

Given two matrices, $\mathbb{A} \in \mathbb{R}^{m \times n}$ and $\mathbb{B} \in \mathbb{R}^{p \times q}$, their Kronecker product is

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{mp \times nq}, \text{ where } c_{k\ell} = a_{i_1 j_1} b_{i_2 j_2}, \tag{1.8}$$

where the relationship between $(k, \ell), (i_1, j_1), (i_2, j_2)$ is as follows, given zero-indexed input indices $(i_1, j_1, i_2, j_2) \in [m] \otimes [n] \otimes [p] \otimes [q]$,

$$k = pi_1 + i_2 \text{ and } \ell = qj_1 + j_2. \tag{1.9}$$

Elementwise, the Kronecker product can be visualized as

$$\begin{bmatrix} a_{11}b_{11} & \cdots & a_{11}b_{1q} & \cdots\cdots & a_{1n}b_{11} & \cdots & a_{1n}b_{1q} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & \cdots & a_{11}b_{pq} & \cdots\cdots & a_{1n}b_{p1} & \cdots & a_{1n}b_{pq} \\ \vdots & & & & & & \vdots \\ \vdots & & & \ddots & & & \vdots \\ \vdots & & & & & & \vdots \\ a_{m1}b_{11} & \cdots & a_{m1}b_{1q} & \cdots\cdots & a_{mn}b_{11} & \cdots & a_{mn}b_{1q} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & \cdots & a_{m1}b_{pq} & \cdots\cdots & a_{mn}b_{p1} & \cdots & a_{mn}b_{pq} \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{bmatrix}.$$

One could also express the Kronecker product in the block format as:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}.$$

Matrix Khatri-Rao Products

Given matrices $\mathbf{A} \in \mathbb{R}^{m\times p}$ and $\mathbf{B} \in \mathbb{R}^{n\times p}$, the Khatri-Rao product computes the columnwise Kronecker product with the columns of its inputs. We define the Khatri-Rao product as

$$\mathbf{C} = \mathbf{A} \odot \mathbf{B} \in \mathbb{R}^{mn\times p}, \text{ where } c_{k\ell} = a_{i\ell}b_{j\ell}, \tag{1.10}$$

and the relationship between the zero-indexed indices $k \in [mn]$ and $(i, j) \in [m] \otimes [n]$ is $k = ni + j$. In terms of the columns of $\mathbf{A}, \mathbf{B}$, we have that $\mathbf{c}_\ell = \mathbf{a}_\ell \otimes \mathbf{b}_\ell$, $\forall \ell \in [p]$, or

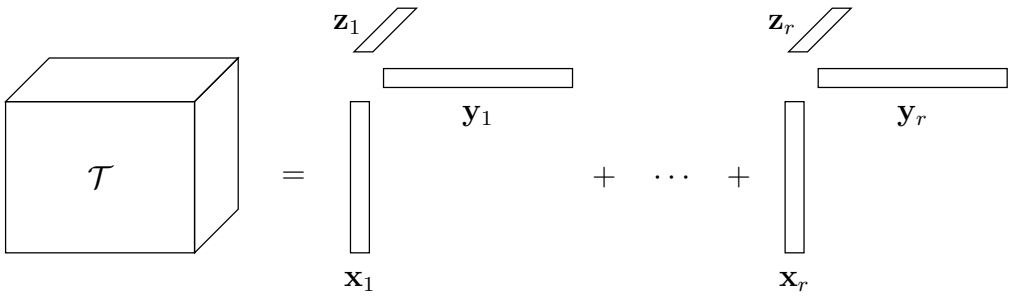
$$\mathbf{C} = \begin{bmatrix} | & & | \\ \mathbf{a}_1 \otimes \mathbf{b}_1 & \cdots & \mathbf{a}_p \otimes \mathbf{b}_p \\ | & & | \end{bmatrix}.$$

1.2 Tensor Decompositions

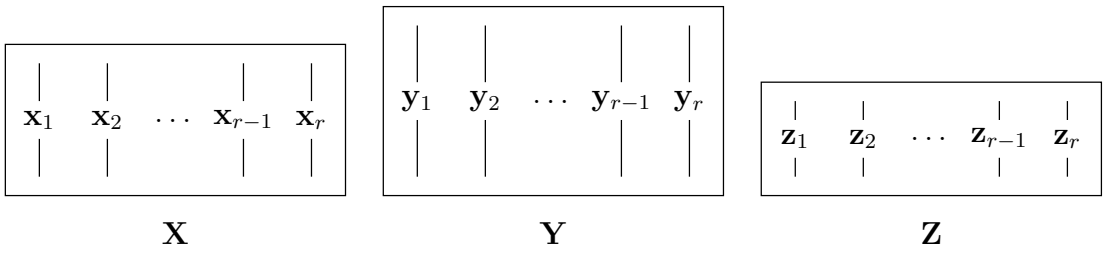
Tensors suffer from the infamous **curse of dimensionality**. This curse exists because as the number of the dimensions of a tensor grows, its storage cost and the cost of operations involving it grows exponentially. This is because the number of entries in a cubical d-way tensor is n^d. Thus, there is often a need to compress these large datasets. **Tensor decompositions** are techniques that decompose tensors into smaller structured representations. Similar to most types of matrix decompositions, we seek a set of matrices/tensors that can be multiplied together appropriately to reconstruct the input. Matrix or tensor decompositions can be either exact or approximate. Most tensor decompositions can be viewed as higher-order generalizations of matrix decompositions. There are several types of tensor decompositions, but in this work we focus on the following two; the CP Decomposition and the Tucker Decomposition. Chapter 2 focuses on exact CP decompositions of a special type of tensor. Chapter 3 focuses on tensor approximations using the Tucker Decomposition. We now briefly introduce these tensor decompositions.

1.2.1 Kruskal Tensors and the CP Decomposition

The Canonical Polyadic (CP) Decomposition compresses an input tensor into a Kruskal Tensor (KTensor), which is a sum of r rank-1 components. Each component is an outer product of r vectors. We refer to r as the rank of the CP decomposition, though this is technically true only when r is minimal. We can visualize this in the case of a 3-way tensor as shown in figure 1.9a. The vectors in each mode are concatenated as columns to form a factor matrix as seen in figure 1.9b. It is crucial to note that the order of the components is arbitrary.



(a) A 3 way Kruskal Tensor Diagram



(b) The vectors of the components of the Kruskal tensor come together to form factor matrices

Figure 1.9: The CP Decomposition

Mathematically, given a tensor $\mathcal{T} \in \mathbb{R}^{m \times n \times p}$ and decomposition rank $r \in \mathbb{N}$, the

goal of an approximate CP Decomposition is to find factor matrices $\mathbf{X} \in \mathbb{R}^{m \times r}, \mathbf{Y} \in \mathbb{R}^{n \times r}, \mathbf{Z} \in \mathbb{R}^{p \times r}$ such that

$$t_{ijk} \approx \sum_{\ell=1}^{r} x_{i\ell} y_{j\ell} z_{k\ell}, \; \forall (i,j,k) \in [m] \times [n] \times [p], \tag{1.11}$$

or alternatively

$$\mathcal{T} \approx [\![\mathbf{X}, \mathbf{Y}, \mathbf{Z}]\!] = \sum_{\ell=1}^{r} x_\ell \circ y_\ell \circ z_\ell. \tag{1.12}$$

The memory footprint of a KTensor is $3rn$. The approximation gets more accurate as r increases. This is the 2D matrix equivalent of having an $n \times n$ matrix approximation to be the sum of the outer product of two $n \times 1$ vectors. Traditional methods of computing the CP decomposition of a tensor through numerical oprimization are gradient descent and Newton's method. The method used in this work is a variation of the latter called damped Gauss-Newton (DGN), which is explained in section 2.1.3. The goal of a CP Decomposition is to minimize the sum of squares errors with the restraint of the user specified rank r as defined as

$$\min \|\mathcal{T} - [\![\mathbf{X}, \mathbf{Y}, \mathbf{Z}]\!]\|^2, \text{ subject to } \mathbf{X} \in \mathbb{R}^{m \times r} \mathbf{Y} \in \mathbb{R}^{n \times r} \mathbf{Z} \in \mathbb{R}^{p \times r}. \tag{1.13}$$

where $\mathcal{T} - [\![\mathbf{X}, \mathbf{Y}, \mathbf{Z}]\!]$ is defined element wise as follows

$$\|\mathcal{T} - [\![\mathbf{X}, \mathbf{Y}, \mathbf{Z}]\!]\|^2 \equiv \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{p} \left(t_{ijk} - \sum_{\ell=1}^{r} x_{i\ell} y_{j\ell} z_{k\ell} \right)^2. \tag{1.14}$$

1.2.2 Tucker Tensors and The Tucker Decomposition

The Tucker Decomposition compresses an input tensor into a Tucker Tensor (TTensor), which is a smaller core tensor with a factor matrix for each of its modes. To reconstruct the approximation of the original tensor, each factor matrix is multiplied with the core in its respective mode through a TTM. We can visualize this in the case of a 3-way tensor as shown in figure 1.10.

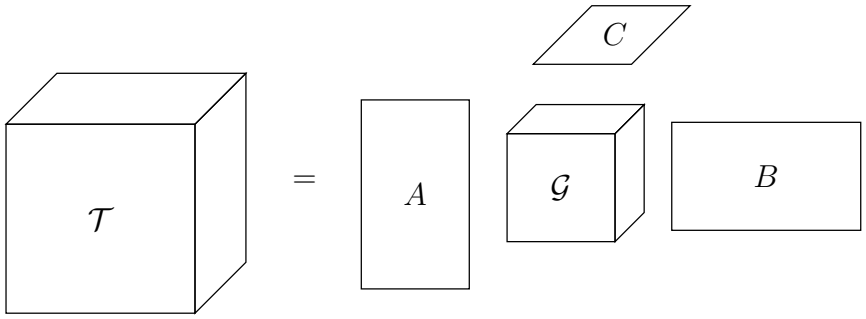


Figure 1.10: A 3-way Tucker Tensor Diagram

If we consider a 3D tensor of size n^3 with core of size r^3 where $r < n$, then the number of entries of a TTensor is $3rn+r^3$ which is less than the original memory footprint and much less if $r \ll n$. The reconstruction of the original tensor is performed using TTMs:

$$\mathcal{T} \approx [\![\mathcal{G}; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!] = \mathcal{G} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} \tag{1.15}$$

and a single entry of the reconstruction can be expressed as

$$t_{ijk} = \sum_{\alpha=1}^{q} \sum_{\beta=1}^{r} \sum_{\gamma=1}^{s} g_{\alpha\beta\gamma} \cdot a_{i\alpha} b_{j\beta} c_{i\gamma}, \forall (i,j,k) \in [m] \otimes [n] \otimes [p] \tag{1.16}$$

The traditional methods for computing a Tucker decomposition re Higher Order Singular Value Decomposition (HOSVD) and Sequentially Truncated Higher Order Singular Value Decomposition (STHOSVD). Though Higher Order Orthogonal Iteration (HOOI) is not as traditional, it will be the focus of chapter 3. We describe these algorithms in section 3.1.

Search For Fast Matrix Multiplication Algorithms

Section 1.1.4 introduced the original matrix multiplication algorithm of $O(n^3)$ cost, yet alternative algorithms exist that achieve the same result with reduced computational complexity. A substantial amount of research has been dedicated to the development of fast matrix multiplication algorithms, motivated by the fact that even the most advanced known algorithms remain far from the theoretical lower bound of $O(n^2 \log n)$. Indeed, the fastest existing algorithms—operating in the range of $O(n^{2.3})$ to $O(n^{2.4})$—are not practical for most applications. As a result, research in this area remains active, with continued efforts to discover more practically efficient algorithms. Although the methods explored in this thesis do not set new benchmarks, they offer meaningful contributions toward that goal.

To properly contextualize the role of CP Decomposition in this work, it is first necessary to review the foundational concepts behind fast matrix multiplication algorithms. Section 2.1.1 begins by laying the groundwork with a introduction to fast matrix multiplication algorithms. Section 2.1.2 summarizes how previous work ex-

plored the application of CP Decompositions to a specific tensor in the search for efficient matrix multiplication algorithms. Our contributions begin in section 2.2 where we explore a structure of fast matrix multiplication algorithms called **cyclic invariance**. We explain such structure in section 2.2.1 and we adapt the ideas of section 2.1.2 to solely explore algorithms that have the cyclic invariance in section 2.2.2. We summarize our findings in section 2.2.3.

Finally, in section 2.3, we outline the future direction of this research. This section is the result of a collaborative effort with the Department of Mathematics at Wake Forest University. With the partnership of Dr. Frank Moore and Dr. Pratyush Mishra, we investigate additional symmetries that fast matrix multiplication algorithms might exhibit and examine whether any of the algorithms identified in this work possess such characteristics.

2.1 Matrix Multiplication Algorithms

2.1.1 Fast Matrix Multiplication Algorithms

Recall from section 1.1.4 the way matrix multiplications are performed. In particular, (1.5) demonstrates how to multiply two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{2\times 2}$. If we interpret the values of $\mathbf{A}$ and $\mathbf{B}$ not as scalars but as submatrices, it becomes apparent that, we can perform such matrix multiplication using a recursive algorithm like the one on algorithm 1, which is given more compactly in figure 2.1.

Algorithm 1 Recursive Matrix Multiplication

function $\mathbf{C} = \textsc{MatMul}(\mathbf{A}, \mathbf{B})$
  if $\dim(\mathbf{A}) = \dim(\mathbf{A}) = 1$ **then**
    return $\mathbf{A} \cdot \mathbf{B}$
  end if
  Divide into quadrants: $\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix}$
  $\mathbf{M}_1 = \text{MatMul}(\mathbf{A_{11}}, \mathbf{B_{11}})$
  $\mathbf{M}_2 = \text{MatMul}(\mathbf{A_{12}}, \mathbf{B_{21}})$
  $\mathbf{M}_3 = \text{MatMul}(\mathbf{A_{11}}, \mathbf{B_{12}})$
  $\mathbf{M}_4 = \text{MatMul}(\mathbf{A_{12}}, \mathbf{B_{22}})$
  $\mathbf{M}_5 = \text{MatMul}(\mathbf{A_{21}}, \mathbf{B_{21}})$
  $\mathbf{M}_6 = \text{MatMul}(\mathbf{A_{22}}, \mathbf{B_{12}})$
  $\mathbf{M}_7 = \text{MatMul}(\mathbf{A_{21}}, \mathbf{B_{12}})$
  $\mathbf{M}_8 = \text{MatMul}(\mathbf{A_{22}}, \mathbf{B_{22}})$
  return $\mathbf{C} = \begin{bmatrix} \mathbf{M}_1 + \mathbf{M}_2 & \mathbf{M}_3 + \mathbf{M}_4 \\ \mathbf{M}_5 + \mathbf{M}_6 & \mathbf{M}_7 + \mathbf{M}_8 \end{bmatrix}$
end function

$$
\begin{aligned}
\mathbf{M}_1 &= \mathbf{A}_{11} \cdot \mathbf{B}_{11} \\
\mathbf{M}_2 &= \mathbf{A}_{12} \cdot \mathbf{B}_{21} \\
\mathbf{M}_3 &= \mathbf{A}_{11} \cdot \mathbf{B}_{12} \\
\mathbf{M}_4 &= \mathbf{A}_{12} \cdot \mathbf{B}_{22} \\
\mathbf{M}_5 &= \mathbf{A}_{21} \cdot \mathbf{B}_{11} \\
\mathbf{M}_6 &= \mathbf{A}_{22} \cdot \mathbf{B}_{21} \\
\mathbf{M}_7 &= \mathbf{A}_{21} \cdot \mathbf{B}_{12} \\
\mathbf{M}_8 &= \mathbf{A}_{22} \cdot \mathbf{B}_{22} \\
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_2 \\
\mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_4 \\
\mathbf{C}_{21} &= \mathbf{M}_5 + \mathbf{M}_6 \\
\mathbf{C}_{22} &= \mathbf{M}_7 + \mathbf{M}_8
\end{aligned}
$$

Figure 2.1: Classic 2 by 2 Matrix Multiplication Algorithm

This algorithm involves 8 multiplications and 4 additions of matrices of size $n/2$. Therefore, the computational complexity is $T(n) = 8T(n/2) + O(n^2) = O(n^{\log_2 8}) = O(n^3)$. This computational cost can be decreased by carefully manipulating our

$$
\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22}) \cdot (\mathbf{B}_{11} + \mathbf{B}_{22}) \\
\mathbf{M}_2 &= (\mathbf{A}_{12} + \mathbf{A}_{22}) \cdot \mathbf{B}_{11} \\
\mathbf{M}_3 &= \mathbf{A}_{11} \cdot (\mathbf{B}_{21} - \mathbf{B}_{22}) \\
\mathbf{M}_4 &= \mathbf{A}_{22} \cdot (\mathbf{B}_{12} - \mathbf{B}_{11}) \\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{21}) \cdot \mathbf{B}_{22} \\
\mathbf{M}_6 &= (\mathbf{A}_{12} - \mathbf{A}_{11}) \cdot (\mathbf{B}_{11} + \mathbf{B}_{21}) \\
\mathbf{M}_7 &= (\mathbf{A}_{21} - \mathbf{A}_{22}) \cdot (\mathbf{B}_{12} + \mathbf{B}_{22}) \\
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{21} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}
$$

Figure 2.2: Classic Strassen's Algorithm

multiplications and additions in order to reduce the number of recursive calls. In 1969, Volker Strassen became the first to develop an algorithm with cost less than $O(n^3)$, which made way for **fast matrix multiplication algorithms**. His classic algorithm, showcased in figure 2.2, involves 7 multiplications. The computational complexity of Strassen's algorithm is $T(n) = 7T(n/2) + O(n^2) = O(n^{\log_2 7}) \approx O(n^{2.81})$.

Strassen's algorithm can be rearranged; we can modify the additions and multiplies to get a permuted version of his algorithm with the same number of multiplies and additions. An example of these variations of Strassen's algorithms can be seen in 2.3. Notice how the two algorithms are the same, except that $\mathbf{M}_3$ became $\mathbf{M}_6$, $\mathbf{M}_4$ became $\mathbf{M}_3$, $\mathbf{M}_6$ became $\mathbf{M}_7$, and $\mathbf{M}_7$ became $\mathbf{M}_4$ and the additions in $\mathbf{C}$ changed respectively. There are multiple ways one could permute Strassen's classic algorithm to obtain other valid fast matrix multiplication algorithms. Furthermore, there are

$$
\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22}) \cdot (\mathbf{B}_{11} + \mathbf{B}_{22}) \\
\mathbf{M}_2 &= (\mathbf{A}_{12} + \mathbf{A}_{22}) \cdot \mathbf{B}_{11} \\
\mathbf{M}_3 &= \mathbf{A}_{22} \cdot (\mathbf{B}_{12} - \mathbf{B}_{11}) \\
\mathbf{M}_4 &= (\mathbf{A}_{21} - \mathbf{A}_{22}) \cdot (\mathbf{B}_{12} + \mathbf{B}_{22}) \\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{21}) \cdot \mathbf{B}_{22} \\
\mathbf{M}_6 &= \mathbf{A}_{11} \cdot (\mathbf{B}_{21} - \mathbf{B}_{22}) \\
\mathbf{M}_7 &= (\mathbf{A}_{12} - \mathbf{A}_{11}) \cdot (\mathbf{B}_{11} + \mathbf{B}_{21}) \\
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_3 + \mathbf{M}_4 - \mathbf{M}_5 \\
\mathbf{C}_{12} &= \mathbf{M}_2 + \mathbf{M}_3 \\
\mathbf{C}_{21} &= \mathbf{M}_5 + \mathbf{M}_6 \\
\mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_6 + \mathbf{M}_7
\end{aligned}
$$

Figure 2.3: Permuted Strassen's Algorithm

other types of transformations that can be applied to Strassen's original algorithm to obtain even more alternative algorithms (see section 2.3). One example is the algorithm presented in figure 2.4, which can no longer be achieved by simply permuting Strassen's classic algorithm. Both algorithms presented in figures 2.3 and 2.4 will be relevant in section 2.2 as they contain a special structure that is hard to visualize in the classic algorithm of figure 2.2. The specific representations of the algorithms of figures 2.3 and 2.4 were chosen because the above structure is easily visualizable in these representations and they contain slighly different variations of the aforementioned structure.

This naturally leads to the question: how many such possibilities exist? We could search for different solutions that all have the same number of multiplications by performing either an exhaustive search or an optimization search on each parameter

$$
\begin{aligned}
\mathbf{M}_1 &= \mathbf{A}_{11} \cdot \mathbf{B}_{11} \\
\mathbf{M}_2 &= (\mathbf{A}_{12} + \mathbf{A}_{22}) \cdot (\mathbf{B}_{12} + \mathbf{B}_{22}) \\
\mathbf{M}_3 &= (\mathbf{A}_{22} - \mathbf{A}_{21}) \cdot (\mathbf{B}_{22} - \mathbf{B}_{21}) \\
\mathbf{M}_4 &= (\mathbf{A}_{21} - \mathbf{A}_{12} - \mathbf{A}_{22}) \cdot (\mathbf{B}_{21} - \mathbf{B}_{12} - \mathbf{B}_{22}) \\
\mathbf{M}_5 &= (-\mathbf{A}_{12}) \cdot (-\mathbf{B}_{21}) \\
\mathbf{M}_6 &= (\mathbf{A}_{11} - \mathbf{A}_{12} + \mathbf{A}_{21} - \mathbf{A}_{22}) \cdot (-\mathbf{B}_{12}) \\
\mathbf{M}_7 &= (-\mathbf{A}_{21}) \cdot (\mathbf{B}_{11} - \mathbf{B}_{12} + \mathbf{B}_{21} - \mathbf{B}_{22}) \\
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_5 \\
\mathbf{C}_{12} &= \mathbf{M}_4 - \mathbf{M}_3 + \mathbf{M}_5 - \mathbf{M}_6 \\
\mathbf{C}_{21} &= \mathbf{M}_2 - \mathbf{M}_4 - \mathbf{M}_5 - \mathbf{M}_7 \\
\mathbf{C}_{22} &= \mathbf{M}_2 + \mathbf{M}_3 - \mathbf{M}_4 - \mathbf{M}_5
\end{aligned}
$$

Figure 2.4: Variant Strassen's Algorithm

involved in matrix multiplication as seen in figure 2.5. If we are searching for a discrete solution with only -1, 0, 1 as coefficients then we have 3^{84} possibilities, though not all of these possibilities would be valid algotithms. We will see how we can decrease this number later on to something much more feasible using the structure hinted at in these algorithms. Before exploring non-exhaustive search strategies for these algorithms, we introduce the central focus of this project.

2.1.2 The Matrix Multiplication Tensor

Central to this work is a specific tensor; the Matrix Multiplication Tensor. It enables matrix multiplication to be represented in tensor form as illustrated in figure 2.6. To carry out the multiplication of matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times p}$, we can form the respective matrix multiplication tensor $\mathcal{M} \in \mathbb{R}^{mn \times np \times mp}$ through Algorithm 2.

$$
\begin{aligned}
\mathbf{M}_1 &= (u_{11}^{(1)}\mathbf{A}_{11} + u_{12}^{(1)}\mathbf{A}_{12} + u_{21}^{(1)}\mathbf{A}_{21} + u_{22}^{(1)}\mathbf{A}_{22}) \cdot (v_{11}^{(1)}\mathbf{B}_{11} + v_{12}^{(1)}\mathbf{B}_{12} + v_{21}^{(1)}\mathbf{B}_{21} + v_{22}^{(1)}\mathbf{B}_{22}) \\
\mathbf{M}_2 &= (u_{11}^{(2)}\mathbf{A}_{11} + u_{12}^{(2)}\mathbf{A}_{12} + u_{21}^{(2)}\mathbf{A}_{21} + u_{22}^{(2)}\mathbf{A}_{22}) \cdot (v_{11}^{(2)}\mathbf{B}_{11} + v_{12}^{(2)}\mathbf{B}_{12} + v_{21}^{(2)}\mathbf{B}_{21} + v_{22}^{(2)}\mathbf{B}_{22}) \\
\mathbf{M}_3 &= (u_{11}^{(3)}\mathbf{A}_{11} + u_{12}^{(3)}\mathbf{A}_{12} + u_{21}^{(3)}\mathbf{A}_{21} + u_{22}^{(3)}\mathbf{A}_{22}) \cdot (v_{11}^{(3)}\mathbf{B}_{11} + v_{12}^{(3)}\mathbf{B}_{12} + v_{21}^{(3)}\mathbf{B}_{21} + v_{22}^{(3)}\mathbf{B}_{22}) \\
\mathbf{M}_4 &= (u_{11}^{(4)}\mathbf{A}_{11} + u_{12}^{(4)}\mathbf{A}_{12} + u_{21}^{(4)}\mathbf{A}_{21} + u_{22}^{(4)}\mathbf{A}_{22}) \cdot (v_{11}^{(4)}\mathbf{B}_{11} + v_{12}^{(4)}\mathbf{B}_{12} + v_{21}^{(4)}\mathbf{B}_{21} + v_{22}^{(4)}\mathbf{B}_{22}) \\
\mathbf{M}_5 &= (u_{11}^{(5)}\mathbf{A}_{11} + u_{12}^{(5)}\mathbf{A}_{12} + u_{21}^{(5)}\mathbf{A}_{21} + u_{22}^{(5)}\mathbf{A}_{22}) \cdot (v_{11}^{(5)}\mathbf{B}_{11} + v_{12}^{(5)}\mathbf{B}_{12} + v_{21}^{(5)}\mathbf{B}_{21} + v_{22}^{(5)}\mathbf{B}_{22}) \\
\mathbf{M}_6 &= (u_{11}^{(6)}\mathbf{A}_{11} + u_{12}^{(6)}\mathbf{A}_{12} + u_{21}^{(6)}\mathbf{A}_{21} + u_{22}^{(6)}\mathbf{A}_{22}) \cdot (v_{11}^{(6)}\mathbf{B}_{11} + v_{12}^{(6)}\mathbf{B}_{12} + v_{21}^{(6)}\mathbf{B}_{21} + v_{22}^{(6)}\mathbf{B}_{22}) \\
\mathbf{M}_7 &= (u_{11}^{(7)}\mathbf{A}_{11} + u_{12}^{(7)}\mathbf{A}_{12} + u_{21}^{(7)}\mathbf{A}_{21} + u_{22}^{(7)}\mathbf{A}_{22}) \cdot (v_{11}^{(7)}\mathbf{B}_{11} + v_{12}^{(7)}\mathbf{B}_{12} + v_{21}^{(7)}\mathbf{B}_{21} + v_{22}^{(7)}\mathbf{B}_{22})
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{C}_{11} &= w_{11}^{(1)}\mathbf{M}_1 + w_{11}^{(2)}\mathbf{M}_2 + w_{11}^{(3)}\mathbf{M}_3 + w_{11}^{(4)}\mathbf{M}_4 + w_{11}^{(5)}\mathbf{M}_5 + w_{11}^{(6)}\mathbf{M}_6 + w_{11}^{(7)}\mathbf{M}_7 \\
\mathbf{C}_{12} &= w_{12}^{(1)}\mathbf{M}_1 + w_{12}^{(2)}\mathbf{M}_2 + w_{12}^{(3)}\mathbf{M}_3 + w_{12}^{(4)}\mathbf{M}_4 + w_{12}^{(5)}\mathbf{M}_5 + w_{12}^{(6)}\mathbf{M}_6 + w_{12}^{(7)}\mathbf{M}_7 \\
\mathbf{C}_{21} &= w_{21}^{(1)}\mathbf{M}_1 + w_{21}^{(2)}\mathbf{M}_2 + w_{21}^{(3)}\mathbf{M}_3 + w_{21}^{(4)}\mathbf{M}_4 + w_{21}^{(5)}\mathbf{M}_5 + w_{21}^{(6)}\mathbf{M}_6 + w_{21}^{(7)}\mathbf{M}_7 \\
\mathbf{C}_{22} &= w_{22}^{(1)}\mathbf{M}_1 + w_{22}^{(2)}\mathbf{M}_2 + w_{22}^{(3)}\mathbf{M}_3 + w_{22}^{(4)}\mathbf{M}_4 + w_{22}^{(5)}\mathbf{M}_5 + w_{22}^{(6)}\mathbf{M}_6 + w_{22}^{(7)}\mathbf{M}_7
\end{aligned}
$$

Figure 2.5: Exhaustive Search of Fast MatMul Algorithms

Algorithm 2 Forming the Matrix Multiplication Tensor

function $\mathcal{T} =$ MatMul-Tensor(m, n, p)
  $\mathcal{T} = \texttt{zeros}(mn, np, mp)$ ▷ Initialize Tensor
  for $i = 1 : m$ **do**
    for $i = 1 : m$ **do**
      for $i = 1 : m$ **do**
        $\mathcal{T}(mj + i, nk + j, pi + k) = 1$
      end for
    end for
  end for
end function

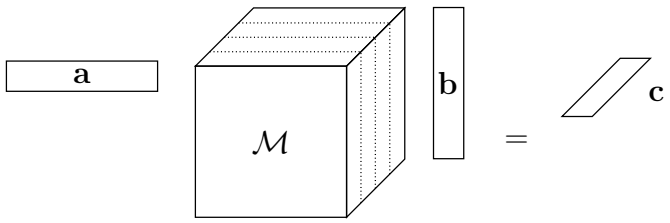


Figure 2.6: Matrix Multiplication in Tensor Format, where $\mathbf{a} = \text{vec}(\mathbf{A})$, $\mathbf{b} = \text{vec}(\mathbf{B})$, and $\mathbf{c} = \text{vec}(\mathbf{C}^\intercal)$.

In order to perform matrix multiplication through the matrix multiplication tensor, we vectorize $\mathbf{A}$ and $\mathbf{B}$ and perform TTMs in the first and second mode respectively. The output, in the third mode, is the vectorization of the output $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ transposed (i.e. $\mathbf{C}^\intercal$). Notation wise this is the same as

$$\mathcal{M} \times_1 vec(\mathbf{A}) \times_2 vec(\mathbf{B}) = \mathcal{M} \times_1 \begin{bmatrix} \mathbf{a}_{11} \\ \mathbf{a}_{12} \\ \mathbf{a}_{21} \\ \mathbf{a}_{22} \end{bmatrix} \times_2 \begin{bmatrix} \mathbf{b}_{11} \\ \mathbf{b}_{12} \\ \mathbf{b}_{21} \\ \mathbf{b}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{c}_{11} \\ \mathbf{c}_{21} \\ \mathbf{c}_{12} \\ \mathbf{c}_{22} \end{bmatrix} = vec(\mathbf{C}^{\mathrm{T}}). \tag{2.1}$$

where $\mathbf{a}_{ij} = \text{vec}\mathbf{A}_{ij}$. This project is concerned only with square matrices, thus we always assume $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{n \times n}$ and $\mathcal{M} \in \mathbb{R}^{n^2 \times n^2 \times n^2}$.

We now revisit the concept of tensor decompositions, specifically the CP decomposition. Recall from 1.2.1 that the decomposition compresses an input tensor into r d-way outer product components. If we decompose the matrix multiplication tensor using the CP Decomposition, there is a hidden fast matrix multiplication algorithm embedded in the components of its CP decomposition. In fact, the number of components r corresponds to the number of multiplications in the algorithm. As a result,

two implications arise. The first implication is that given an algorithm, say one of the 3^{84} for 2×2 algorithms with rank 7, we can form the KTensor of the corresponding algorithm, namely $\hat{\mathcal{M}}$ and take the norm of the difference from the original matrix multiplication tensor. If $\|\mathcal{M} - \hat{\mathcal{M}}\| = 0$, then the algorithm in the factor matrices of $\hat{\mathcal{M}}$ is a valid fast matrix multiplication algorithm. Before continuing with the second implication, we must understand how to visualize the hidden algorithm in the factor matrices of a KTensor.

Figure 2.7 shows Strassen's original algorithm with two representations, the one we have seen before on the left and the KTensor representation in figure 2.7b. The way to interpret the algorithm on the right, is that each horizontal lines separate the factor matrices, therefore just like in figure 1.9a the factor matrices $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ are separated by the horizontal lines, the columns of the factor matrices represent the rank-one components. A careful examination reveals how the columns representing $\mathbf{M}_\ell$ correspond to the representation on the right. If a 1 or -1 appear in the right representation then they appear as $\mathbf{A}_{ij}$ or $-\mathbf{A}_{ij}$ respectively on the left.

The other implication is that we can decompose the matrix multiplication tensor with specified CP rank (as in (1.13)) through an optimization algorithm to search for fast matrix multiplication algorithms.

$$
\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22}) \cdot (\mathbf{B}_{11} + \mathbf{B}_{22}) \\
\mathbf{M}_2 &= (\mathbf{A}_{12} + \mathbf{A}_{22}) \cdot \mathbf{B}_{11} \\
\mathbf{M}_3 &= \mathbf{A}_{11} \cdot (\mathbf{B}_{21} - \mathbf{B}_{22}) \\
\mathbf{M}_4 &= \mathbf{A}_{22} \cdot (\mathbf{B}_{12} - \mathbf{B}_{11}) \\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{21}) \cdot \mathbf{B}_{22} \\
\mathbf{M}_6 &= (\mathbf{A}_{12} - \mathbf{A}_{11}) \cdot (\mathbf{B}_{11} + \mathbf{B}_{21}) \\
\mathbf{M}_7 &= (\mathbf{A}_{21} - \mathbf{A}_{22}) \cdot (\mathbf{B}_{12} + \mathbf{B}_{22}) \\
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{21} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}
$$

(a) Recursive Algorithm Format

	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$	$\mathbf{M}_4$	$\mathbf{M}_5$	$\mathbf{M}_6$	$\mathbf{M}_7$
$\mathbf{A}_{11}$	1	0	1	0	1	−1	0
$\mathbf{A}_{12}$	0	1	0	0	0	1	0
$\mathbf{A}_{21}$	0	0	0	0	1	0	1
$\mathbf{A}_{22}$	1	1	0	1	0	0	−1
$\mathbf{B}_{11}$	1	1	0	−1	0	1	0
$\mathbf{B}_{12}$	0	0	0	1	0	0	1
$\mathbf{B}_{21}$	0	0	1	0	0	1	0
$\mathbf{B}_{22}$	1	0	−1	0	1	0	1
$\mathbf{C}_{11}$	1	0	0	1	−1	0	1
$\mathbf{C}_{21}$	0	0	1	0	1	0	0
$\mathbf{C}_{12}$	0	1	0	1	0	0	0
$\mathbf{C}_{22}$	1	−1	1	0	0	1	0

(b) KTensor Format

Figure 2.7: A comparison of classic Strassen's algorithm in regular and KTensor formats. The horizontal lines indicate the factor matrices of the KTensor

2.1.3 Damped Gauss Newton Optimization for CP Decompositions

To search for fast matrix multiplication algorithms, we employ numerical optimization techniques to solve the CP decomposition problem formulated in (1.13). This subsection is dedicated to outlining and deriving the steps and tools required to construct an algorithm tailored specifically to this problem. As a starting point, we define the input to the optimization as the vectorization of the KTensor, given by:

$$
\mathbf{v} = \operatorname{vec}\left(\begin{bmatrix} \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \end{bmatrix}\right) = \begin{bmatrix} \operatorname{vec}(\mathbf{X}) \\ \operatorname{vec}(\mathbf{Y}) \\ \operatorname{vec}(\mathbf{Z}) \end{bmatrix} \in \mathbb{R}^{3nr}.
$$

Using $\mathbf{v}$ as the input to our unconstrained optimization problem seen in (1.14), we obain the following representation of the same equation

$$\min f(\mathbf{v}) = \frac{1}{2}\|\phi(\mathbf{v})\|^2 : \mathbb{R}^{3nr} \to \mathbb{R}, \tag{2.2}$$

where we define the nonlinear function

$$\phi(\mathbf{v}) = \text{vec}\left(\mathcal{M} - [\![\mathbf{X}, \mathbf{Y}, \mathbf{Z}]\!]\right) : \mathbb{R}^{3nr} \to \mathbb{R}^{n^3}. \tag{2.3}$$

It is important to emphasize that $[\![\mathbf{X}, \mathbf{Y}, \mathbf{Z}]\!]$ denotes the 3-way tensor constructed from the factor matrices of the KTensor, which is obtained from $\mathbf{v}$. Since our objective is to find exact solutions, it is advantageous to use an algorithm that converges rapidly in the neighborhood of a solution to the optimization problem of (2.2). The classic Newton algorithm is a natural choice due to its quadratic convergence. The search direction in Newton's method is computed by solving Newton's equation

$$\nabla^2 f(\mathbf{v})\mathbf{d}_k = -\nabla f(\mathbf{v}),$$

where $\nabla^2 f(\mathbf{v})$ is the Hessian, $\nabla f(\mathbf{v})$ is the gradient, and $\mathbf{d}_k$ is our search direction defined as

$$\mathbf{d}_k = \text{vec}\left(\begin{bmatrix} \bar{\mathbf{X}} \\ \bar{\mathbf{Y}} \\ \bar{\mathbf{Z}} \end{bmatrix}\right) = \begin{bmatrix} \text{vec}(\bar{\mathbf{X}}) \\ \text{vec}(\bar{\mathbf{Y}}) \\ \text{vec}(\bar{\mathbf{Z}}) \end{bmatrix} \in \mathbb{R}^{3nr}.$$

The use of Newton's method requires evaluating the Hessian matrix, $\nabla^2 f(\mathbf{v})$, which can be computationally expensive and difficult to derive—especially given the

cyclic invariant structure discussed in section 2.2. As an alternative, the Gauss-Newton method offers a practical approximation of the Hessian with $\mathbf{J}^\intercal\mathbf{J}$, where $\mathbf{J} : \mathbb{R}^n \to \mathbb{R}^{n\times m}$ is the Jacobian of the function $\phi(\mathbf{v})$. The Gauss-Newton equation is defined as

$$(\mathbf{J}^\intercal\mathbf{J})\mathbf{d}_k = -\nabla f(\mathbf{v})$$

.

However, a limitation of the Gauss-Newton method is that the approximation $\mathbf{J}^\intercal\mathbf{J}$ is often singular. The damped Gauss-Newton (DGN) method adds a damping parameter, λ, to enforce positive definiteness. Thus, the DGN equation is defined as

$$(\mathbf{J}^\intercal\mathbf{J} + \lambda\mathbf{I})\mathbf{d}_k = -\nabla f(\mathbf{v}). \tag{2.4}$$

The goal of the remainder of this section is to show the equations necessary to implement the DGN for the CP decomposition (CP_DGN) method. We omit the details of the derivation of such equations as it goes beyond the scope of this work. The interested reader is referred to [1]. We begin with the right-hand side of the DGN equation. The gradient is

$$\nabla f = \text{vec}\left(\begin{bmatrix} \partial f/\partial \mathbf{X} \\ \partial f/\partial \mathbf{Y} \\ \partial f/\partial \mathbf{Z} \end{bmatrix}\right) = \begin{bmatrix} \partial f/\partial \text{vec}(\mathbf{X}) \\ \partial f/\partial \text{vec}(\mathbf{Y}) \\ \partial f/\partial \text{vec}(\mathbf{Z}) \end{bmatrix} \in \mathbb{R}^{3n^2r}$$

where each subequation is defined as

$$\begin{aligned} \partial f/\partial \mathbf{X} &= -\mathbf{M}_{(1)}(\mathbf{Z} \odot \mathbf{Y}) + \mathbf{X}(\mathbf{Z}^\intercal \mathbf{Z} * \mathbf{Y}^\intercal \mathbf{Y}) \in \mathbb{R}^{n^2 \times r} \\ \partial f/\partial \mathbf{Y} &= -\mathbf{M}_{(2)}(\mathbf{Z} \odot \mathbf{X}) + \mathbf{Y}(\mathbf{Z}^\intercal \mathbf{Z} * \mathbf{X}^\intercal \mathbf{X}) \in \mathbb{R}^{n^2 \times r} \\ \partial f/\partial \mathbf{Z} &= -\mathbf{M}_{(3)}(\mathbf{Y} \odot \mathbf{X}) + \mathbf{Z}(\mathbf{Y}^\intercal \mathbf{Y} * \mathbf{X}^\intercal \mathbf{X}) \in \mathbb{R}^{n^2 \times r} \end{aligned} \tag{2.5}$$

On the other hand, we have the Jacobian, which is defined as

$$\mathbf{J} = \begin{bmatrix} \mathbf{J_A} & \mathbf{J_B} & \mathbf{J_C} \end{bmatrix} \in \mathbb{R}^{n^3 \times 3nr} \tag{2.6}$$

where

$$\begin{aligned} \mathbf{J_X} &\equiv \partial \phi/\partial \mathrm{vec}(\mathbf{X}) = (\mathbf{Z} \odot \mathbf{Y}) \otimes \mathbf{I} \in \mathbb{R}^{n^3 \times nr} \\ \mathbf{J_Y} &\equiv \partial \phi/\partial \mathrm{vec}(\mathbf{Y}) = \mathbf{\Pi}_2^\intercal (\mathbf{Z} \odot \mathbf{X}) \otimes \mathbf{I} \in \mathbb{R}^{n^3 \times nr} \\ \mathbf{J_Z} &\equiv \partial \phi/\partial \mathrm{vec}(\mathbf{Z}) = \mathbf{\Pi}_3^\intercal (\mathbf{Y} \odot \mathbf{X}) \otimes \mathbf{I} \in \mathbb{R}^{n^3 \times nr} \end{aligned} \tag{2.7}$$

such that $\mathbf{\Pi}_k$ is the tensor perfect shuffle matrix ([1], Definition 2.24), such that $\mathrm{vec}(\mathcal{P})$ $= \Pi_k \ \mathrm{vec}(\mathbf{P}_{(k)})$, and $\mathbf{\Pi}_1$ is not written explicitly because it is the identity matrix. We consider fast application of $\mathbf{J}^\intercal \mathbf{J} + \lambda \mathbf{I}$, rather than forming $\mathbf{J}$ or $\mathbf{J}^\intercal \mathbf{J}$ explicitly; we use the structure of these matrices to compute the matrix-vector product $(\mathbf{J}^\intercal \mathbf{J} + \lambda \mathbf{I})\mathbf{d}_k$ without computing any explicit Kronecker or Khatri-Rao products. From (2.6) we have that the block structure of $\mathbf{J}^\intercal \mathbf{J}$ is

$$\mathbf{J}^\intercal \mathbf{J} = \begin{bmatrix} \mathbf{J_X^\intercal J_X} & \mathbf{J_X^\intercal J_Y} & \mathbf{J_X^\intercal J_Z} \\ \mathbf{J_Y^\intercal J_X} & \mathbf{J_Y^\intercal J_Y} & \mathbf{J_Y^\intercal J_Z} \\ \mathbf{J_Z^\intercal J_X} & \mathbf{J_Z^\intercal J_Y} & \mathbf{J_Z^\intercal J_Z} \end{bmatrix}$$

By distributing the left-hand side of the DGN equation, we obtain $\mathbf{J}^\intercal \mathbf{J}\mathbf{d}_k + \lambda \mathbf{I}\mathbf{d}_k$. While the second term is trivial to derive, the first term becomes

$$\mathbf{J}^\intercal\mathbf{J}\mathbf{d}_k = \begin{bmatrix} \mathbf{J}_\mathbf{X}^\intercal\mathbf{J}_\mathbf{X}\text{vec}(\bar{\mathbf{X}}) + \mathbf{J}_\mathbf{X}^\intercal\mathbf{J}_\mathbf{Y}\text{vec}(\bar{\mathbf{Y}}) + \mathbf{J}_\mathbf{X}^\intercal\mathbf{J}_\mathbf{Z}\text{vec}(\bar{\mathbf{Z}}) \\ \mathbf{J}_\mathbf{Y}^\intercal\mathbf{J}_\mathbf{X}\text{vec}(\bar{\mathbf{X}}) + \mathbf{J}_\mathbf{Y}^\intercal\mathbf{J}_\mathbf{Y}\text{vec}(\bar{\mathbf{Y}}) + \mathbf{J}_\mathbf{Y}^\intercal\mathbf{J}_\mathbf{Z}\text{vec}(\bar{\mathbf{Z}}) \\ \mathbf{J}_\mathbf{Z}^\intercal\mathbf{J}_\mathbf{X}\text{vec}(\bar{\mathbf{X}}) + \mathbf{J}_\mathbf{Z}^\intercal\mathbf{J}_\mathbf{Y}\text{vec}(\bar{\mathbf{Y}}) + \mathbf{J}_\mathbf{Z}^\intercal\mathbf{J}_\mathbf{Z}\text{vec}(\bar{\mathbf{Z}}) \end{bmatrix}. \tag{2.8}$$

With some careful algebra ([1], Proposition 13.1), it can be shown (2.8) leads to

$$\mathbf{J}^\intercal\mathbf{J}\mathbf{d}_k = \begin{bmatrix} \text{vec}(\bar{\mathbf{X}}(\mathbf{Y}^\intercal\mathbf{Y} * \mathbf{Z}^\intercal\mathbf{Z}) + \mathbf{X}(\bar{\mathbf{Y}}^\intercal\mathbf{Y} * \mathbf{Z}^\intercal\mathbf{Z}) + \mathbf{X}(\mathbf{Y}^\intercal\mathbf{Y} * \bar{\mathbf{Z}}^\intercal\mathbf{Z})) \\ \text{vec}(\mathbf{Y}(\bar{\mathbf{X}}^\intercal\mathbf{X} * \mathbf{Z}^\intercal\mathbf{Z}) + \bar{\mathbf{Y}}(\mathbf{X}^\intercal\mathbf{X} * \mathbf{Z}^\intercal\mathbf{Z}) + \mathbf{Y}(\mathbf{X}^\intercal\mathbf{X} * \bar{\mathbf{Z}}^\intercal\mathbf{Z})) \\ \text{vec}(\mathbf{Z}(\bar{\mathbf{X}}^\intercal\mathbf{X} * \mathbf{Y}^\intercal\mathbf{Y}) + \mathbf{Z}(\mathbf{X}^\intercal\mathbf{X} * \bar{\mathbf{Y}}^\intercal\mathbf{Y}) + \bar{\mathbf{Z}}(\mathbf{X}^\intercal\mathbf{X} * \mathbf{Y}^\intercal\mathbf{Y})) \end{bmatrix}. \tag{2.9}$$

We now have all of the necssary tools to create the algorithm that searches for fast matrix multiplication algorithms through the CP decomposition of the matrix multiplication tensor. This algorithm is showcased in detail in algorithm 3. A couple important observations must be made. First, note that we do not solve the DGN equation equation (2.4) directly. Rather in line 11 we use a conjugate gradient iterative algorithm to solve the DGN equation, which is the reason it is sufficient to be able to apply $(\mathbf{J}^\intercal\mathbf{J} + \lambda\mathbf{I})$ to $\mathbf{d}_k$ instead of computing the approximate Hessian of $\phi(\mathbf{v})$ explicitly. Secondly, we update $\mathbf{d}_k$ using backtracking line search using the Goldstein Conditions, for details see ([1], B.3.1).

2.2 Cyclic Invariance

We can reduce our search space in our search for fast matrix multiplication algorithms by levereging **cyclic invariance**. Cyclic invariance is an added structure in

Algorithm 3 Damped Gauss-Newton On The Matrix Multiplication Tensor

Input: Matrix Multiplication Tensor $\mathcal{M}$,
  CP Tensor Rank r,
  Damping Parameter $\lambda \in \mathbb{R}^+$,
  Convergence Tolerance $\epsilon > 0$
Output: CP Tensor $\mathcal{K}$
function DGN($\mathcal{M}, r, \lambda, \epsilon$)
  Initialize K and $\mathrm{K}_{\mathrm{prev}}$ to be a cell of length 3 of $n^2 \times r$ matrices
  for $i = 1 : MaxIters$ **do**
    f $\longleftarrow \frac{1}{2}\|\mathcal{M} - \mathcal{K}\|^2$ ▷ Compute Function Value
    $\nabla \mathbf{f} \longleftarrow [\mathrm{vec}\left(\frac{\partial f}{\partial \mathbf{X}}\right) \mathrm{vec}\left(\frac{\partial f}{\partial \mathbf{Y}}\right) \mathrm{vec}\left(\frac{\partial f}{\partial \mathbf{Z}}\right)]^\intercal$ ▷ Compute Gradient
    $S \longleftarrow$ Solution to $(\mathbf{J}^\intercal\mathbf{J} + \lambda\mathbf{I})K = -\nabla\mathbf{f}$ ▷ Conjugate Gradient Iter. Alg.
    while *Goldstein Conditions Are Not Satisfied* **do**
      $\mathrm{K} \longleftarrow \mathrm{K}_{\mathrm{prev}} + \alpha S$
      $\mathrm{f}_{new} \longleftarrow \frac{1}{2}\|\mathcal{M} - \mathcal{K}\|^2$
      $\alpha \longleftarrow \alpha/2$
    end while
    if f - $\mathrm{f}_{new} < \epsilon$ **then**
      break
    end if
  end for
end function

matrix multiplication algorithms that reduces the number of variables of the CP Decomposition optimization problem for the matrix multiplication tensor by a factor of three.

2.2.1 Cyclic Invariant Matrix Multiplication Algorithms

Recall that we can permute Strassen's agorithms to obtain variations. Some of these variations can be cyclic invariant. Recall from section 2.1.1 that we introduced a variation of Strassen's algorithm in figure 2.3. Figure 2.8 shows both Strassen's original algorithm on the left, and the permuted version in KTensor format in figure 2.8b. Notice how: (1) the permutation is given by simply moving the columns of the KTensor

(in unity) just the way it was described previously. $\mathbf{M}_3$ became $\mathbf{M}_6$ and so on, and (2) the permuted Strassen's algorithm on the right is composed of smaller submatrices that appear throughout the main factor matrices of the KTensor (highlighted in colors). The 4×1 matrix in red is called the **symmetric component**, and we place it at the beginning of all three factor matrices. We denote it as $\mathbf{S}$. The remaining three 4×2 submatrices are called the **cyclic component**, we denote them as $\mathbf{U}, \mathbf{V}, \mathbf{W}$.

	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$	$\mathbf{M}_4$	$\mathbf{M}_5$	$\mathbf{M}_6$	$\mathbf{M}_7$
$\mathbf{A}_{11}$	1	0	1	0	1	−1	0
$\mathbf{A}_{12}$	0	1	0	0	0	1	0
$\mathbf{A}_{21}$	0	0	0	0	1	0	1
$\mathbf{A}_{22}$	1	1	0	1	0	0	−1
$\mathbf{B}_{11}$	1	1	0	−1	0	1	0
$\mathbf{B}_{12}$	0	0	0	1	0	0	1
$\mathbf{B}_{21}$	0	0	1	0	0	1	0
$\mathbf{B}_{22}$	1	0	−1	0	1	0	1
$\mathbf{C}_{11}$	1	0	0	1	−1	0	1
$\mathbf{C}_{21}$	0	0	1	0	1	0	0
$\mathbf{C}_{12}$	0	1	0	1	0	0	0
$\mathbf{C}_{22}$	1	−1	1	0	0	1	0

(a) Classic Strassen's Algoritm from figure 2.2 in KTensor format

	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$	$\mathbf{M}_4$	$\mathbf{M}_5$	$\mathbf{M}_6$	$\mathbf{M}_7$
$\mathbf{A}_{11}$	1	0	0	0	1	1	−1
$\mathbf{A}_{12}$	0	1	0	0	0	0	1
$\mathbf{A}_{21}$	0	0	0	1	1	0	0
$\mathbf{A}_{22}$	1	1	1	−1	0	0	0
$\mathbf{B}_{11}$	1	1	−1	0	0	0	1
$\mathbf{B}_{12}$	0	0	1	1	0	0	0
$\mathbf{B}_{21}$	0	0	0	0	0	1	1
$\mathbf{B}_{22}$	1	0	0	1	1	−1	0
$\mathbf{C}_{11}$	1	0	1	1	−1	0	0
$\mathbf{C}_{21}$	0	0	0	0	1	1	0
$\mathbf{C}_{12}$	0	1	1	0	0	0	0
$\mathbf{C}_{22}$	1	−1	0	0	0	1	1

(b) Permuted Strassen's Algoritm from figure 2.3 in KTensor format

Figure 2.8: Cyclic Invariance in Strassen's Algorithm

Because they are always submatrices of the factor matrices, both the symmetric and the cyclic components have the same number of rows as the factor matrices,

namely n. However, they can have a different number of columns. We denote the number of columns of the symmetric component as r_s and the number of columns of the cyclic component as r_c. Since r is the rank of the CP Decomposition, we have that $r_s + 3r_c = r$. Because of this, given a matrix multiplication tensor of $n \times n$ matrices, and a given rank r, there are multiple choices for r_c, which in turn define the value of r_s since $r_s = r - 3r_c$.

	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$	$\mathbf{M}_4$	$\mathbf{M}_5$	$\mathbf{M}_6$	$\mathbf{M}_7$
$\mathbf{A}_{11}$	1	0	0	0	1	1	−1
$\mathbf{A}_{12}$	0	1	0	0	0	0	1
$\mathbf{A}_{21}$	0	0	0	1	1	0	0
$\mathbf{A}_{22}$	1	1	1	−1	0	0	0
$\mathbf{B}_{11}$	1	1	−1	0	0	0	1
$\mathbf{B}_{12}$	0	0	1	1	0	0	0
$\mathbf{B}_{21}$	0	0	0	0	0	1	1
$\mathbf{B}_{22}$	1	0	0	1	1	−1	0
$\mathbf{C}_{11}$	1	0	1	1	−1	0	0
$\mathbf{C}_{21}$	0	0	0	0	1	1	0
$\mathbf{C}_{12}$	0	1	1	0	0	0	0
$\mathbf{C}_{22}$	1	−1	0	0	0	1	1

(a) Permuted Strassen's Algoritm figure 2.3 in KTensor format

	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$	$\mathbf{M}_4$	$\mathbf{M}_5$	$\mathbf{M}_6$	$\mathbf{M}_7$
$\mathbf{A}_{11}$	1	0	0	0	0	1	0
$\mathbf{A}_{12}$	0	0	−1	1	0	1	−1
$\mathbf{A}_{21}$	0	1	0	−1	−1	−1	0
$\mathbf{A}_{22}$	0	1	1	−1	0	−1	0
$\mathbf{B}_{11}$	1	0	0	0	0	0	1
$\mathbf{B}_{12}$	0	0	−1	1	−1	0	1
$\mathbf{B}_{21}$	0	1	0	−1	0	−1	−1
$\mathbf{B}_{22}$	0	1	1	−1	0	0	−1
$\mathbf{C}_{11}$	1	0	0	0	1	0	0
$\mathbf{C}_{21}$	0	0	−1	1	1	−1	0
$\mathbf{C}_{12}$	0	1	0	−1	−1	0	−1
$\mathbf{C}_{22}$	0	1	1	−1	−1	0	0

(b) Variant Strassen's Algoritm figure 2.4 in KTensor format

Figure 2.9: Different types of Cyclic Invariance in Strassen's Algorithm

For rank 7 algorithms of 2×2 matrices, we have two options $[r_s = 1, r_c = 2]$ and $[r_s = 4, r_c = 1]$. We have seen versions of both of these selected values before in section 2.1.1. The algorithm found in figure 2.3 is a cyclic invariant algorithm with

$r_s = 1, r_c = 2$, and similarly the one in figure 2.4 has $r_s = 4, r_c = 1$. Figure 2.9 shows these algorithms in their KTensor format where their cyclic invariant structure is easily visualizable.

We can visualize imposing the cyclic invariant structure on the factor matrices of the KTensor as transforming figure 1.9b into figure 2.10, and transforming figure 1.9a into figure 2.11.

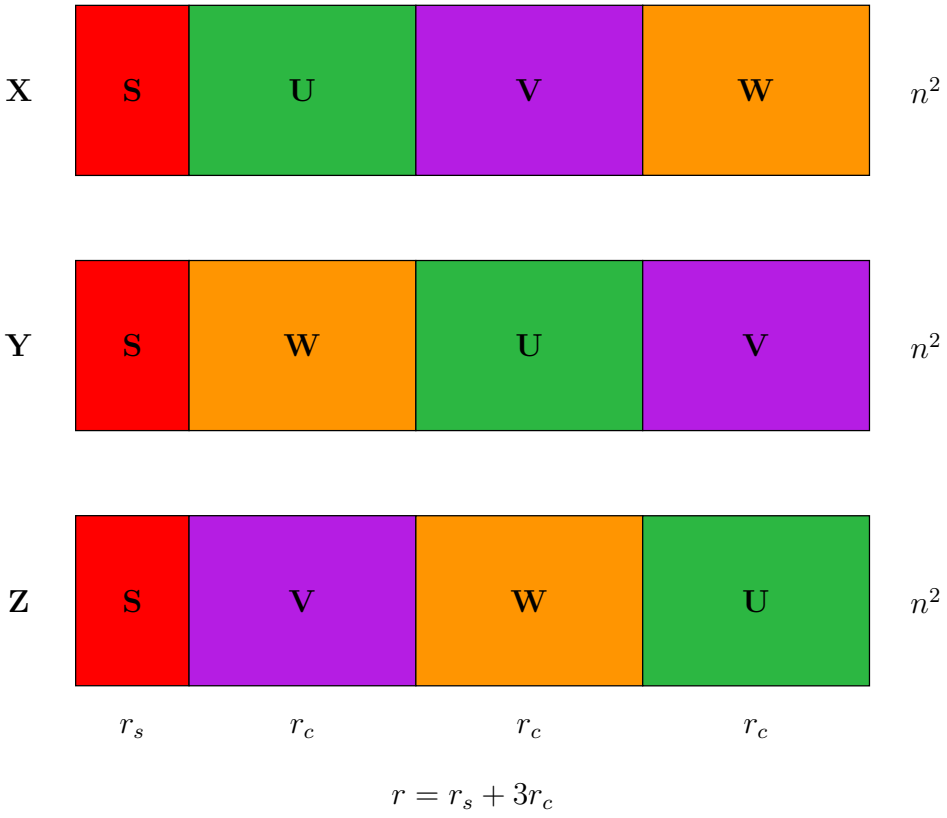


Figure 2.10: Cyclic Invariance in a KTensor

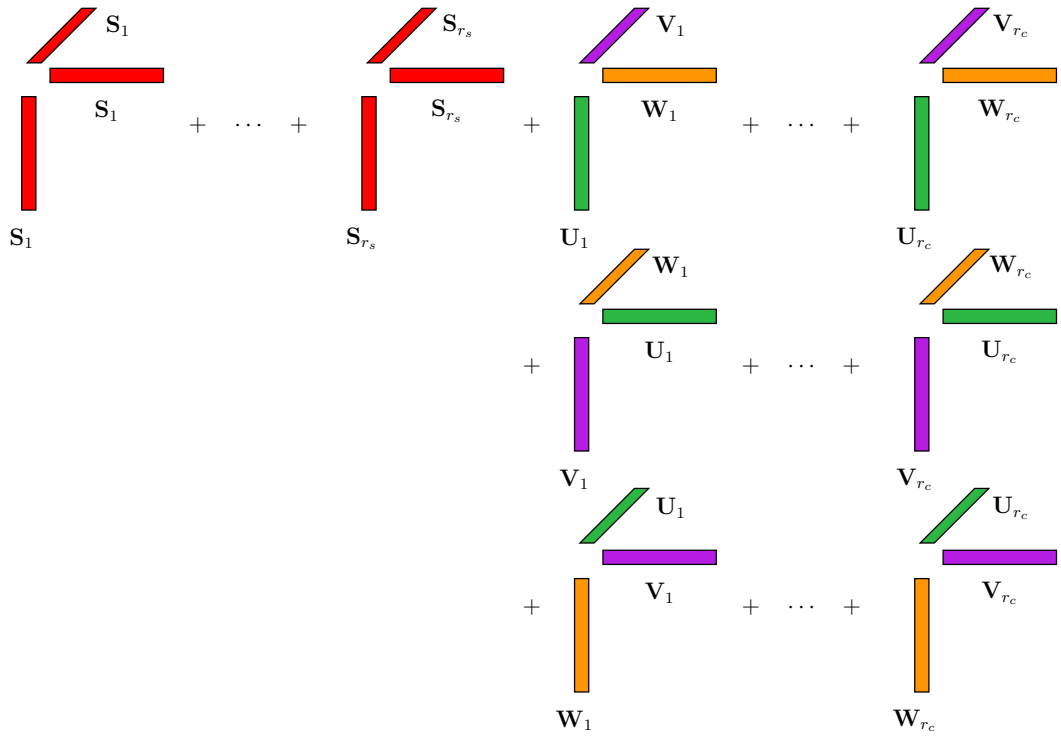


Figure 2.11: CP Decomposition Diagram with Cyclic Invariant Structure

2.2.2 Adapting CP_DGN to Cyclic Invariance

The goal of this section is to adapt algorithm 3 to search for algorithms with cyclic invariant structure. The process of solving the DGN equation $(\mathbf{J}^\intercal\mathbf{J} + \lambda\mathbf{I})\mathbf{d}_k = -\nabla f(\mathbf{v})$ stays the same, but now we adapt $\mathbf{v}$. To exploit the cyclic invariance structure on a KTensor with matrices $\mathbf{X}, \mathbf{Y},$ and $\mathbf{Z}$ we impose that these are built of smaller submatrices $\mathbf{S}, \mathbf{U}, \mathbf{V},$ and $\mathbf{W}$ as in

$$\begin{aligned} \mathbf{X} &= [\ \mathbf{S}\ \ \mathbf{U}\ \ \mathbf{V}\ \ \mathbf{W}\ ] \\ \mathbf{Y} &= [\ \mathbf{S}\ \ \mathbf{W}\ \ \mathbf{U}\ \ \mathbf{V}\ ] \\ \mathbf{Z} &= [\ \mathbf{S}\ \ \mathbf{V}\ \ \mathbf{W}\ \ \mathbf{U}\ ] \end{aligned} \tag{2.10}$$

Therefore, instead of searching for matrices $\mathbf{X}, \mathbf{Y},$ and $\mathbf{Z}$, we are solely searching for matrices $\mathbf{S}, \mathbf{U}, \mathbf{V},$ and $\mathbf{W}$ which are known to be the building blocks of $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$. Essentially, we decreased the number of variables of our optimization problem by a factor of three. If we were to do an exhaustive search like in figure 2.5, we decrease from 3^{84} possibilities to 3^{28} for the 2 by 2 case with 7 multiplications. Our input $\mathbf{v}$ is now defined as

$$\mathbf{v} = \text{vec}\left(\begin{bmatrix} \mathbf{S} \\ \mathbf{U} \\ \mathbf{V} \\ \mathbf{W} \end{bmatrix}\right) = \begin{bmatrix} \text{vec}(\mathbf{S}) \\ \text{vec}(\mathbf{U}) \\ \text{vec}(\mathbf{V}) \\ \text{vec}(\mathbf{W}) \end{bmatrix} \in \mathbb{R}^{nr},$$

Similarly, the search direction $\mathbf{d}_k$ becomes

$$\mathbf{d}_k = \text{vec}\left(\begin{bmatrix} \bar{\mathbf{S}} \\ \bar{\mathbf{U}} \\ \bar{\mathbf{V}} \\ \bar{\mathbf{W}} \end{bmatrix}\right) = \begin{bmatrix} \text{vec}(\bar{\mathbf{S}}) \\ \text{vec}(\bar{\mathbf{U}}) \\ \text{vec}(\bar{\mathbf{V}}) \\ \text{vec}(\bar{\mathbf{W}}) \end{bmatrix} \in \mathbb{R}^{nr}.$$

The function value $f(\mathbf{v})$ to this problem remains the same, we are solely adapting (1.14) to be

$$\|\mathcal{T} - [\![\mathbf{S}, \mathbf{S}, \mathbf{S}]\!] - [\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!] - [\![\mathbf{W}, \mathbf{U}, \mathbf{V}]\!] - [\![\mathbf{V}, \mathbf{W}, \mathbf{U}]\!]\|^2, \tag{2.11}$$

which is equivalent to

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\left(m_{ijk} - \sum_{q}^{r_s} s_{iq}s_{jq}s_{kq} - \sum_{l}^{r_c}(u_{il}v_{jl}w_{kl} + w_{il}u_{jl}v_{kl} + v_{il}w_{jl}u_{kl})\right)^2. \tag{2.12}$$

By decreasing the number of variables, we have increase the complexity of the

equations for the gradient and Jacobian. The equations for the gradient become

$$\nabla f = \begin{bmatrix} \mathsf{vec}(\partial f/\partial \mathbf{S}) \\ \mathsf{vec}(\partial f/\partial \mathbf{U}) \\ \mathsf{vec}(\partial f/\partial \mathbf{V}) \\ \mathsf{vec}(\partial f/\partial \mathbf{W}) \end{bmatrix} \in \mathbb{R}^{nr} \tag{2.13}$$

where each partial derivative is defined as:

$$\begin{aligned}
\partial f/\partial \mathbf{S} &= 3 \cdot \big(\mathbf{S}(\mathbf{S}^\intercal \mathbf{S} * \mathbf{S}^\intercal \mathbf{S}) + \mathbf{U}(\mathbf{V}^\intercal \mathbf{S} * \mathbf{W}^\intercal \mathbf{S}) + \mathbf{V}(\mathbf{U}^\intercal \mathbf{S} * \mathbf{W}^\intercal \mathbf{S}) + \mathbf{W}(\mathbf{U}^\intercal \mathbf{S} * \mathbf{V}^\intercal \mathbf{S}) \\
&\quad - (\mathbf{X}_{(1)} + \mathbf{X}_{(2)} + \mathbf{X}_{(3)})(\mathbf{S} \odot \mathbf{S}) \big) \\
\partial f/\partial \mathbf{U} &= 3 \cdot \big(\mathbf{S}(\mathbf{S}^\intercal \mathbf{V} * \mathbf{S}^\intercal \mathbf{W}) + \mathbf{U}(\mathbf{V}^\intercal \mathbf{V} * \mathbf{W}^\intercal \mathbf{W}) + \mathbf{V}(\mathbf{W}^\intercal \mathbf{V} * \mathbf{U}^\intercal \mathbf{W}) + \mathbf{W}(\mathbf{U}^\intercal \mathbf{V} * \mathbf{V}^\intercal \mathbf{W}) \big) \\
&\quad - \mathbf{X}_{(1)}(\mathbf{V} \odot \mathbf{W}) - \mathbf{X}_{(2)}(\mathbf{W} \odot \mathbf{V}) - \mathbf{X}_{(3)}(\mathbf{V} \odot \mathbf{W}) \\
\partial f/\partial \mathbf{V} &= 3 \cdot \big(\mathbf{S}(\mathbf{S}^\intercal \mathbf{U} * \mathbf{S}^\intercal \mathbf{W}) + \mathbf{U}(\mathbf{W}^\intercal \mathbf{U} * \mathbf{V}^\intercal \mathbf{W}) + \mathbf{V}(\mathbf{U}^\intercal \mathbf{U} * \mathbf{W}^\intercal \mathbf{W}) + \mathbf{W}(\mathbf{V}^\intercal \mathbf{U} * \mathbf{U}^\intercal \mathbf{W}) \big) \\
&\quad - \mathbf{X}_{(1)}(\mathbf{W} \odot \mathbf{U}) - \mathbf{X}_{(2)}(\mathbf{U} \odot \mathbf{W}) - \mathbf{X}_{(3)}(\mathbf{W} \odot \mathbf{U}) \\
\partial f/\partial \mathbf{W} &= 3 \cdot \big(\mathbf{S}(\mathbf{S}^\intercal \mathbf{U} * \mathbf{S}^\intercal \mathbf{V}) + \mathbf{U}(\mathbf{V}^\intercal \mathbf{U} * \mathbf{W}^\intercal \mathbf{V}) + \mathbf{V}(\mathbf{W}^\intercal \mathbf{U} * \mathbf{U}^\intercal \mathbf{V}) + \mathbf{W}(\mathbf{U}^\intercal \mathbf{U} * \mathbf{V}^\intercal \mathbf{V}) \big) \\
&\quad - \mathbf{X}_{(1)}(\mathbf{U} \odot \mathbf{V}) - \mathbf{X}_{(2)}(\mathbf{V} \odot \mathbf{U}) - \mathbf{X}_{(3)}(\mathbf{U} \odot \mathbf{V})
\end{aligned} \tag{2.14}$$

Similarly, our Jacobian becomes

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_\mathbf{S} & \mathbf{J}_\mathbf{U} & \mathbf{J}_\mathbf{V} & \mathbf{J}_\mathbf{W} \end{bmatrix} \in \mathbb{R}^{n^3 \times nr} \tag{2.15}$$

where

$$\begin{aligned}
\mathbf{J}_\mathbf{S} &= (\mathbf{S} \odot \mathbf{S}) \otimes \mathbf{I} + \mathbf{\Pi}_2^\intercal \cdot (\mathbf{S} \odot \mathbf{S}) \otimes \mathbf{I} + \mathbf{\Pi}_3^\intercal \cdot (\mathbf{S} \odot \mathbf{S}) \otimes \mathbf{I} \\
\mathbf{J}_\mathbf{U} &= (\mathbf{V} \odot \mathbf{W}) \otimes \mathbf{I} + \mathbf{\Pi}_2^\intercal \cdot (\mathbf{W} \odot \mathbf{V}) \otimes \mathbf{I} + \mathbf{\Pi}_3^\intercal \cdot (\mathbf{V} \odot \mathbf{W}) \otimes \mathbf{I} \\
\mathbf{J}_\mathbf{V} &= (\mathbf{W} \odot \mathbf{U}) \otimes \mathbf{I} + \mathbf{\Pi}_2^\intercal \cdot (\mathbf{U} \odot \mathbf{W}) \otimes \mathbf{I} + \mathbf{\Pi}_3^\intercal \cdot (\mathbf{W} \odot \mathbf{U}) \otimes \mathbf{I} \\
\mathbf{J}_\mathbf{V} &= (\mathbf{U} \odot \mathbf{V}) \otimes \mathbf{I} + \mathbf{\Pi}_2^\intercal \cdot (\mathbf{V} \odot \mathbf{U}) \otimes \mathbf{I} + \mathbf{\Pi}_3^\intercal \cdot (\mathbf{U} \odot \mathbf{V}) \otimes \mathbf{I}
\end{aligned} \tag{2.16}$$

However, recall that we are not interested in the explicit expressions for the Jacobian, but rather how to apply $(\mathbf{J}^\intercal \mathbf{J} + \lambda \mathbf{I})\mathbf{d}_k$ to our search direction $\mathbf{d}_k$. Just as in section 2.1.2, we ignore the expression for $\lambda \mathbf{I} \mathbf{d}_k$ as it is trivial.

$$\mathbf{J}^\intercal\mathbf{J}\mathbf{d}_k = \begin{bmatrix} \mathbf{J}_\mathbf{S}^\intercal\mathbf{J}_\mathbf{S} & \mathbf{J}_\mathbf{S}^\intercal\mathbf{J}_\mathbf{U} & \mathbf{J}_\mathbf{S}^\intercal\mathbf{J}_\mathbf{V} & \mathbf{J}_\mathbf{S}^\intercal\mathbf{J}_\mathbf{W} \\ \mathbf{J}_\mathbf{U}^\intercal\mathbf{J}_\mathbf{S} & \mathbf{J}_\mathbf{U}^\intercal\mathbf{J}_\mathbf{U} & \mathbf{J}_\mathbf{U}^\intercal\mathbf{J}_\mathbf{V} & \mathbf{J}_\mathbf{U}^\intercal\mathbf{J}_\mathbf{W} \\ \mathbf{J}_\mathbf{V}^\intercal\mathbf{J}_\mathbf{S} & \mathbf{J}_\mathbf{V}^\intercal\mathbf{J}_\mathbf{U} & \mathbf{J}_\mathbf{V}^\intercal\mathbf{J}_\mathbf{V} & \mathbf{J}_\mathbf{V}^\intercal\mathbf{J}_\mathbf{W} \\ \mathbf{J}_\mathbf{W}^\intercal\mathbf{J}_\mathbf{S} & \mathbf{J}_\mathbf{W}^\intercal\mathbf{J}_\mathbf{U} & \mathbf{J}_\mathbf{W}^\intercal\mathbf{J}_\mathbf{V} & \mathbf{J}_\mathbf{W}^\intercal\mathbf{J}_\mathbf{W} \end{bmatrix} \begin{bmatrix} \mathrm{vec}(\bar{\mathbf{S}}) \\ \mathrm{vec}(\bar{\mathbf{U}}) \\ \mathrm{vec}(\bar{\mathbf{V}}) \\ \mathrm{vec}(\bar{\mathbf{W}}) \end{bmatrix} \tag{2.17}$$

The equations for each entry of the above matrix-vector product are described below:

$$\begin{aligned}
\mathbf{J}_\mathbf{S}^\intercal\mathbf{J}_\mathbf{S}\mathrm{vec}(\bar{\mathbf{S}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{S}}(\mathbf{S}^\intercal\mathbf{S})*(\mathbf{S}^\intercal\mathbf{S})+\mathbf{2}\cdot\mathbf{S}(\bar{\mathbf{S}}^\intercal\mathbf{S})*(\mathbf{S}^\intercal\mathbf{S})\Big) \\
\mathbf{J}_\mathbf{S}^\intercal\mathbf{J}_\mathbf{U}\mathrm{vec}(\bar{\mathbf{U}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{U}}(\mathbf{V}^\intercal\mathbf{S})*(\mathbf{W}^\intercal\mathbf{S})+\mathbf{V}(\bar{\mathbf{U}}^\intercal\mathbf{S})*(\mathbf{W}^\intercal\mathbf{S})+\mathbf{W}(\bar{\mathbf{U}}^\intercal\mathbf{S})*(\mathbf{V}^\intercal\mathbf{S})\Big) \\
\mathbf{J}_\mathbf{S}^\intercal\mathbf{J}_\mathbf{V}\mathrm{vec}(\bar{\mathbf{V}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{V}}(\mathbf{U}^\intercal\mathbf{S})*(\mathbf{W}^\intercal\mathbf{S})+\mathbf{U}(\bar{\mathbf{V}}^\intercal\mathbf{S})*(\mathbf{W}^\intercal\mathbf{S})+\mathbf{W}(\bar{\mathbf{V}}^\intercal\mathbf{S})*(\mathbf{U}^\intercal\mathbf{S})\Big) \\
\mathbf{J}_\mathbf{S}^\intercal\mathbf{J}_\mathbf{W}\mathrm{vec}(\bar{\mathbf{W}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{W}}(\mathbf{U}^\intercal\mathbf{S})*(\mathbf{V}^\intercal\mathbf{S})+\mathbf{U}(\bar{\mathbf{W}}^\intercal\mathbf{S})*(\mathbf{V}^\intercal\mathbf{S})+\mathbf{V}(\bar{\mathbf{W}}^\intercal\mathbf{S})*(\mathbf{U}^\intercal\mathbf{S})\Big)
\end{aligned}$$

$$\begin{aligned}
\mathbf{J}_\mathbf{U}^\intercal\mathbf{J}_\mathbf{S}\mathrm{vec}(\bar{\mathbf{S}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{S}}(\mathbf{S}^\intercal\mathbf{V})*(\mathbf{S}^\intercal\mathbf{W})+\mathbf{S}\Big((\bar{\mathbf{S}}^\intercal\mathbf{W})*(\mathbf{S}^\intercal\mathbf{V})+(\bar{\mathbf{S}}^\intercal\mathbf{V})*(\mathbf{S}^\intercal\mathbf{W})\Big)\Big) \\
\mathbf{J}_\mathbf{U}^\intercal\mathbf{J}_\mathbf{U}\mathrm{vec}(\bar{\mathbf{U}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{U}}(\mathbf{V}^\intercal\mathbf{V})*(\mathbf{W}^\intercal\mathbf{W})+\mathbf{V}(\bar{\mathbf{U}}^\intercal\mathbf{W})*(\mathbf{W}^\intercal\mathbf{V})+\mathbf{W}(\bar{\mathbf{U}}^\intercal\mathbf{V})*(\mathbf{V}^\intercal\mathbf{W})\Big) \\
\mathbf{J}_\mathbf{U}^\intercal\mathbf{J}_\mathbf{V}\mathrm{vec}(\bar{\mathbf{V}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{V}}(\mathbf{W}^\intercal\mathbf{V})*(\mathbf{U}^\intercal\mathbf{W})+\mathbf{W}(\bar{\mathbf{V}}^\intercal\mathbf{W})*(\mathbf{U}^\intercal\mathbf{V})+\mathbf{U}(\bar{\mathbf{V}}^\intercal\mathbf{V})*(\mathbf{W}^\intercal\mathbf{W})\Big) \\
\mathbf{J}_\mathbf{U}^\intercal\mathbf{J}_\mathbf{W}\mathrm{vec}(\bar{\mathbf{W}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{W}}(\mathbf{U}^\intercal\mathbf{V})*(\mathbf{V}^\intercal\mathbf{W})+\mathbf{U}(\bar{\mathbf{W}}^\intercal\mathbf{W})*(\mathbf{V}^\intercal\mathbf{V})+\mathbf{V}(\bar{\mathbf{W}}^\intercal\mathbf{V})*(\mathbf{U}^\intercal\mathbf{W})\Big)
\end{aligned}$$

$$\begin{aligned}
\mathbf{J}_\mathbf{V}^\intercal\mathbf{J}_\mathbf{S}\mathrm{vec}(\bar{\mathbf{S}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{S}}(\mathbf{S}^\intercal\mathbf{U})*(\mathbf{S}^\intercal\mathbf{W})+\mathbf{S}\Big((\bar{\mathbf{S}}^\intercal\mathbf{U})*(\mathbf{S}^\intercal\mathbf{W})+(\bar{\mathbf{S}}^\intercal\mathbf{W})*(\mathbf{S}^\intercal\mathbf{U})\Big)\Big) \\
\mathbf{J}_\mathbf{V}^\intercal\mathbf{J}_\mathbf{U}\mathrm{vec}(\bar{\mathbf{U}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{U}}(\mathbf{V}^\intercal\mathbf{W})*(\mathbf{W}^\intercal\mathbf{U})+\mathbf{V}(\bar{\mathbf{U}}^\intercal\mathbf{U})*(\mathbf{W}^\intercal\mathbf{W})+\mathbf{W}(\bar{\mathbf{U}}^\intercal\mathbf{W})*(\mathbf{V}^\intercal\mathbf{U})\Big) \\
\mathbf{J}_\mathbf{V}^\intercal\mathbf{J}_\mathbf{V}\mathrm{vec}(\bar{\mathbf{V}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{V}}(\mathbf{W}^\intercal\mathbf{W})*(\mathbf{U}^\intercal\mathbf{U})+\mathbf{W}(\bar{\mathbf{V}}^\intercal\mathbf{U})*(\mathbf{U}^\intercal\mathbf{W})+\mathbf{U}(\bar{\mathbf{V}}^\intercal\mathbf{W})*(\mathbf{W}^\intercal\mathbf{U})\Big) \\
\mathbf{J}_\mathbf{V}^\intercal\mathbf{J}_\mathbf{W}\mathrm{vec}(\bar{\mathbf{W}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{W}}(\mathbf{U}^\intercal\mathbf{W})*(\mathbf{V}^\intercal\mathbf{U})+\mathbf{U}(\bar{\mathbf{W}}^\intercal\mathbf{U})*(\mathbf{V}^\intercal\mathbf{W})+\mathbf{V}(\bar{\mathbf{W}}^\intercal\mathbf{W})*(\mathbf{U}^\intercal\mathbf{U})\Big)
\end{aligned}$$

$$\begin{aligned}
\mathbf{J}_\mathbf{W}^\intercal\mathbf{J}_\mathbf{S}\mathrm{vec}(\bar{\mathbf{S}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{S}}(\mathbf{S}^\intercal\mathbf{U})*(\mathbf{S}^\intercal\mathbf{V})+\mathbf{S}\Big((\bar{\mathbf{S}}^\intercal\mathbf{U})*(\mathbf{S}^\intercal\mathbf{V})+(\bar{\mathbf{S}}^\intercal\mathbf{V})*(\mathbf{S}^\intercal\mathbf{U})\Big)\Big) \\
\mathbf{J}_\mathbf{W}^\intercal\mathbf{J}_\mathbf{U}\mathrm{vec}(\bar{\mathbf{U}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{U}}(\mathbf{V}^\intercal\mathbf{U})*(\mathbf{W}^\intercal\mathbf{V})+\mathbf{V}(\bar{\mathbf{U}}^\intercal\mathbf{V})*(\mathbf{W}^\intercal\mathbf{U})+\mathbf{W}(\bar{\mathbf{U}}^\intercal\mathbf{U})*(\mathbf{V}^\intercal\mathbf{V})\Big) \\
\mathbf{J}_\mathbf{W}^\intercal\mathbf{J}_\mathbf{V}\mathrm{vec}(\bar{\mathbf{V}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{V}}(\mathbf{W}^\intercal\mathbf{W})*(\mathbf{U}^\intercal\mathbf{V})+\mathbf{W}(\bar{\mathbf{V}}^\intercal\mathbf{V})*(\mathbf{U}^\intercal\mathbf{U})+\mathbf{U}(\bar{\mathbf{V}}^\intercal\mathbf{U})*(\mathbf{W}^\intercal\mathbf{V})\Big) \\
\mathbf{J}_\mathbf{W}^\intercal\mathbf{J}_\mathbf{W}\mathrm{vec}(\bar{\mathbf{W}}) &= 3\cdot\mathrm{vec}\Big(\bar{\mathbf{W}}(\mathbf{U}^\intercal\mathbf{U})*(\mathbf{V}^\intercal\mathbf{V})+\mathbf{U}(\bar{\mathbf{W}}^\intercal\mathbf{V})*(\mathbf{V}^\intercal\mathbf{U})+\mathbf{V}(\bar{\mathbf{W}}^\intercal\mathbf{U})*(\mathbf{U}^\intercal\mathbf{V})\Big).
\end{aligned}$$

With all of these modifications, we introduce a modified version of the CP_DGN algorithm that searches for fast matrix multiplication algorithms with cyclic invariant structure. Details can be found in algorithm 4.

Algorithm 4 Cyclic Invariant CP Damped Gauss-Newton

1 **Input:** Matrix Multiplication Tensor $\mathcal{M}$,
2 CP Tensor Rank r,
3 Damping Parameter $\lambda \in \mathbb{R}$,
4 Convergence Tolerance $\epsilon > 0$
5 **Output:** CP Tensor $\mathcal{K}$
6 **function** DGN($\mathcal{M}, r, \lambda, \epsilon$)
7 Initialize K and K_prev to be a cell of length 4 with the first entry being of $n^2 \times r_s$ and the remaining three $n^2 \times r_c$ matrices
8 **for** $i = 1 : MaxIters$ **do**
9 f $\longleftarrow \frac{1}{2}\|\mathcal{M} - \mathcal{K}\|^2$ ▷ Solution to equation (2.12)
10 $\nabla \mathbf{f} \longleftarrow [\text{vec}\left(\frac{\partial f}{\partial \mathbf{S}}\right) \text{vec}\left(\frac{\partial f}{\partial \mathbf{U}}\right) \text{vec}\left(\frac{\partial f}{\partial \mathbf{V}}\right) \text{vec}\left(\frac{\partial f}{\partial \mathbf{W}}\right)]^T$
11 $S \longleftarrow$ Solution to $(\mathbf{J}^T\mathbf{J} + \lambda I)K = -\nabla \mathbf{f}$
12 **while** *Goldstein Conditions Are Not Satisfied* **do**
13 K $\longleftarrow$ K_prev + αS
14 f $_{new} \longleftarrow$ Compute Function Value
15 $\alpha \longleftarrow \alpha/2$
16 **end while**
17 **if** f - f $_{new} < \epsilon$ **then**
18 **break**
19 **end if**
20 **end for**
21 **end function**

2.2.3 Heuristics and Our Findings

This section describes the heuristics of the process of searching for fast matrix multiplication algorithms as well as a summary of the identified algorithms using algorithm 4 for selected values of n, r, r_s, and r_c.

Given n and r we search for all possible combinations of r_s and r_c. For each combination, we perform a CP decompositon using algorithm 4 with thousands of random starts to initialize initial KTensor in line 7. We perform our experiments on a server so that we can take advantage of multiple processors on these random starts.

After we obtain an approximation of the matrix multiplication tensor (i.e. after a full passing of algorithm 4), we run through CI_CP_DGN again, but instead of a random start, we use the output of the previous passing as a starting point. However, in between iterations, we modify the solution of the previous output. First, we sparsify our solution by applying a transformation that maintains the approximation but introduces some zeroes into a single component (see section 2.3). This transformation tends to also introduce more zeroes in other components. Secondly, we also force the coefficients to be between -1 and 1 before initializing further rounds of CI_CP_DGN. We save any solution, at any point, that results in an exact decomposition of the matrix multiplication tensor. With our testing scheme being defined, we discuss our findings.

For $n = 2$, it is well-established that no solutions exist for $r < 7$. Nonetheless, for $r = 7$ it is noteworthy that several solutions were found for $r_s = 1$ and $r_s = 4$, although many of these are equivalent up to permutations. For the case of $n = 3$, the best-known algorithm corresponds to $r = 23$, while the theoretical lower bound stands at $r = 19$. We explored values $r = 22, 21, 20$, and 19, but our algorithm did not yield any novel results at these ranks. At $r = 23$, all three r_s values identified corresponded to previously known solutions. We were also unable to find new algorithms for $r_s = 8, 14, 17$, and 20.

Our first novel results emerge in the case of $n = 4$ algorithms. While the values

$n = 2, r = 4$	S	$n = 3, r = 23$	S	$n = 4, r = 49$	S	$n = 5$	S
$r_s = 1, r_c = 2$	1000s	$r_s = 2, r_c = 7$	100s	$r_s = 1, r_c = 16$	6	$r = 109$	2
$r_s = 4, r_c = 1$	1000s	$r_s = 5, r_c = 6$	100s	$r_s = 4, r_c = 15$	0	$r = 93$	0
		$r_s = 8, r_c = 5$	-	$r_s = 7, r_c = 14$	-	$r = 91$	0
		$r_s = 11, r_c = 4$	100s	$r_s = 10, r_c = 13$	-	$\cdots$	-
		$r_s = 14, r_c = 3$	-	$r_s = 13, r_c = 12$	2		
		$r_s = 17, r_c = 2$	-	$r_s = 16, r_c = 11$	32		
		$r_s = 20, r_c = 1$	-	$\cdots$	-		

Table 2.1: A Summary of our found Algorithms discovered using CI_CP_DGN. S corresponds to the number of solutions found given using the specified symmetric and cyclic ranks. An approximation is given for $n = 2$ and $n = 3$ as too many algorithms were found and there is not a record of exactly how many were found. Details are omitted from $n = 5$ as an extensive search is yet to be performed.

$r_s = 1$ and $r_s = 16$ correspond to known solutions—being equivalent to applying Strassen's algorithm twice—our method successfully identified a previously unkown fast matrix multiplication algorithm with $r_s = 13$. Unfortunately, despite the existence of algorithms with $r_s = 4$, our approach did not yield any such results, which is likely due to limitations in the search process which could mean further explorations could reveal an algorithm in this case. No algorithms were found for $r_s = 7$, 10, or any $r_s \geq 19$. Our exploration for $n = 5$ was limited, and a more rigorous search remains to be conducted.

2.3 Further Structure in Matrix Multiplication Algorithms

Throughout this work, we have explored the cyclic invariance structure in fast matrix multiplication algorithms and how it reduces the number of variables in CP decomposition of the matrix multiplication tensor [2, 3, 4]. There are two other types of symmetries not explored in this work, namely the the GL_n^3 and the *transpose* symmetries.

We can express the cyclic transformation as a map on a CP decomposition of the matrix multiplication tensor:

$$[\![\mathbf{X}, \mathbf{Y}, \mathbf{Z}]\!] \to [\![\mathbf{Z}, \mathbf{X}, \mathbf{Y}]\!]. \tag{2.18}$$

The output of a cyclic transformation is always also a valid matrix multiplication algorithm; if the transformation produces the same components, just in a different order, then we say that the algorithm is cyclic invariant. Similarly, we define the transpose transformation as the following action on a CP decomposition

$$[\![\mathbf{X}, \mathbf{Y}, \mathbf{Z}]\!] \to [\![\mathbf{\Pi_{n\times n}} \mathbf{Y}, \mathbf{\Pi_{n\times n}} \mathbf{X}, \mathbf{\Pi_{n\times n}} \mathbf{Z}]\!], \tag{2.19}$$

where $\mathbf{\Pi}_{n\times n}$ is the (matrix) perfect shuffle permuation matrix that satisfies $\mathbf{\Pi}_{n\times n}\text{vec}(\mathbf{M}) = \text{vec}(\mathbf{M}^\intercal)$ [5]. As before, this transformation also results in another valid matrix multiplication algorithm. This property exists because $\mathbf{AB} = \mathbf{C}$ implies $\mathbf{A}^\intercal\mathbf{B}^\intercal = \mathbf{C}^\intercal$. Lastly, the GL_n^3 action is defined as an action that tranforms a CP

decomposition

$$[\![\mathbf{X}, \mathbf{Y}, \mathbf{Z}]\!] \to [\![(\mathbf{Q}^{-\intercal} \otimes \mathbf{P})\mathbf{X}, (\mathbf{R}^{-\intercal} \otimes \mathbf{Q})\mathbf{Y}, (\mathbf{P}^{-\intercal} \otimes \mathbf{R})\mathbf{Z}]\!], \tag{2.20}$$

where $\mathbf{P}, \mathbf{Q},$ and $\mathbf{R}$ are any $n \times n$ nonsingular matrices. The reason this also results in another valid algorithm is because $\mathbf{AB} = \mathbf{C}$ implies $(\mathbf{PAQ})(\mathbf{Q^{-1}BR}) = \mathbf{PCR}$. The special case where $\mathbf{Q} = \mathbf{P} = \mathbf{R}$ is called the diagonal action, which maintains the cyclic invariance structure of an algorithm if it exists. If the output of a transpose action or the GL_n^3 action is the same as the input, possibly with the components reordered, then we say the algorithm is invariant to that action.

We refer to the set of transformations that can map a given tensor decomposition to itself (up to permutations of the components of the decomposition) as the symmetry group of the algorithm. Currently, we are studying the algorithms for $n = 4$ $r = 49$ case that we have found. Our goal is to identify the symmetry groups of these algorithms, to find any additional invariance ((2.19) or (2.20)) in these algorithms. Prior work has done this manually, but we are attempting to automate this process. Future work is to introduce the corresponding structures while searching for these algorithms. For example, imposing the cyclic invariant structure in the search for fast matrix multiplication algorithms reduced the number of variables by a factor of three. Similarly, imposing the transpose structure reduces the number of variables by a factor of 2. Imposing both simultaneously reduces it by a factor of 6.

Parallel Rank-Adaptive HOOI

Previous work has shown that the Tucker decomposition is particularly effective at compressing datasets arising from scientific simulations occurring in two or three spatial dimensions and through time, in part because algorithms for computing the Tucker decomposition can scale to high performance computing platforms (see, e.g., [6, 7, 8, 9, 10, 11, 12]). When used as a technique for compression, the Tucker format has an advantage that subtensors can be efficiently decompressed without reconstructing the full tensor, which allows for fast visualization of particular time steps, spatial regions, or quantities of interest. The Tucker decomposition is a generalization of the truncated singular value decomposition (SVD) that consists of a core tensor, with as many modes as the input, and a set of factor matrices. The dimensions of the core tensor are known as the Tucker ranks, and like the truncated SVD, smaller ranks yield higher compression but larger error, in contrast, larger ranks yield lower compression but smaller error. This is known as the compression-accuracy trade-off as seen in Figure 3.1.

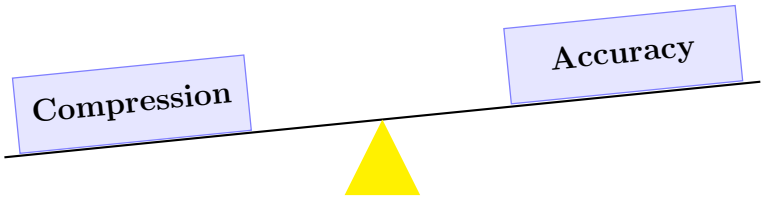


Figure 3.1: The Tensor Decomposition Trade-Off

To specify compression beforehand, we constrain the size of the core tensor $\mathcal{G}$. In the 3-way case, we specify the array of ranks $\mathbf{r} = [q, r, s]$, or in the d-way case, $\mathbf{r} = [r_1, \ldots, r_d]$. Because we set the ranks beforehand, this is called the **rank-specified** formulation, and with it, we also know the compression ratio beforehand which for the 3-way case is

$$\frac{mnp}{qrs + qm + nr + sp} \approx \frac{mnp}{qrs}. \tag{3.1}$$

In this formulation, we cannot determine in advance what the accuracy will be. On the other hand, to specify accuracy beforehand, we constrain the maximum relative error threshold ϵ of the Tucker approximation. The 3-way case of the relative error can is defined as

$$\frac{\|\mathcal{X} - [\![\mathcal{G}; \mathbf{U}, \mathbf{V}, \mathbf{W}]\!]\|}{\|\mathcal{X}\|} \leq \epsilon. \tag{3.2}$$

This is called the **error-specified** formulation, where we cannot determine in advance what the compression will be.

As we describe in section 3.1, a direct algorithm known as Sequentially Truncated Higher Order SVD (ST-HOSVD) achieves quasi-optimal accuracy among decompositions of specified ranks, and it can adaptively determine ranks to solve the

error-specified formulation [13, 14]. The Higher Order Orthogonal Iteration (HOOI) algorithm is an iterative method that solves the rank-specified formulation of the problem [15, 16, 17]. Conventional wisdom has held that because ST-HOSVD solves the rank-specified problem to within a small factor of the optimal solution, HOOI is useful only to refine ST-HOSVD's solution and is typically unnecessary [18, 9, 19]. Based on the observations that (1) a single iteration of HOOI is computationally cheaper than ST-HOSVD, particularly when the compression ratio is high, and (2) when initialized randomly, HOOI tends to converge to a solution as accurate as that of ST-HOSVD in as few as one or two iterations, the goal of this work is to evaluate the scalability of HOOI to large tensor datasets and compare its performance with state-of-the-art implementations of ST-HOSVD.

One of the main limitations of HOOI is that it solves the rank-specified formulation of the Tucker approximation problem, but it does not solve the error-specified formulation. In section 3.1.5, we propose a rank-adaptive variant of HOOI that does solve the error-specified formulation. Our approach is based on incrementally expanding the Tucker ranks over HOOI iterations in order to satisfy the error threshold and then, once it is satisfied, truncating the ranks to maximize compression. We exploit fast computation of the approximation error of a given Tucker approximation and all its leading subtensors to determine the best truncation. Thus, prior knowledge of the output ranks is no longer required, but the choice of initial ranks affects the number

of HOOI iterations performed.

TuckerMPI is a C++/MPI library that implements ST-HOSVD for large dense tensors [9]. We build our parallelization of HOOI on TuckerMPI, leveraging the existing functionality for the main computational kernels required of both ST-HOSVD and HOOI, including the TTM computation and algorithms for computing the SVD. The efficiency and scalability of HOOI is largely determined by those of the TTM and SVD kernels. We apply two key optimizations, one for each kernel, in order to make our rank-adaptive parallel HOOI algorithm more efficient. To reduce the computational costs of the TTM kernel, we use memoization to avoid recomputation of individual TTMs that occur across subiterations of HOOI; see section 3.1.3. To reduce costs and expose better parallelism of SVD computations, we use subspace iteration within HOOI subiterations. While subspace iteration computes only an approximation to the leading left singular vectors, we show that one subspace iteration is sufficient to obtain the desired accuracy across the full HOOI iteration. Implementation of subspace iteration requires new parallel computational kernels in TuckerMPI, which we describe in section 3.1.4.

In section 3.3, we evaluate the efficiency and scalability of HOOI and compare it to TuckerMPI's ST-HOSVD. We consider synthetic test data to show how the number of modes and the compression ratios affect performance, and we demonstrate the impact of our computational optimizations in different scenarios for the rank-specified

approximation problem. We also consider three real datasets generated from scientific simulation of fluid flow and combustion to test the rank-adaptivity of our algorithm. The experimental results demonstrate that HOOI generally scales as well as ST-HOSVD. In cases of large tensor dimension, ST-HOSVD becomes bottlenecked by a sequential SVD-related computation, and HOOI scales significantly better than ST-HOSVD at high core counts. We show that HOOI benefits from the reduction of computational cost, roughly proportional to the compression ratio in a single tensor dimension, compared to ST-HOSVD, but that it can suffer from lower local kernel efficiency as a result. For scenarios of high compression ratio and initial ranks that are overestimates of the output ranks, we observe that HOOI achieves Tucker approximations faster than ST-HOSVD, and in many cases, produces Tucker decompositions with better compression ratio.

3.1 Tucker Algorithms

Recall from Section 1.2.2 that a Tucker decomposition of a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ approximates $\mathcal{X}$ as a product of a core tensor $\mathcal{G} \in \mathbb{R}^{r_1 \cdots r_d}$ and factor matrices $\mathbf{U}_k \in \mathbb{R}^{n_k \times r_k} \forall k \in [d]$ where $\mathcal{X} \approx \hat{\mathcal{X}} = \mathcal{G} \times_1 \mathbf{U}_1 \cdots \times \mathbf{U}_d$. The optimal rank-$\mathbf{r}$ Tucker decomposition of $\mathcal{X}$ can be expressed as a solution to the rank-specified optimization problem

$$\begin{aligned} \min \quad & \|\mathcal{X} - (\mathcal{G} \times_1 \mathbf{U}_1 \cdots \times \mathbf{U}_d)\| \\ \text{subject to } & \mathcal{G} \in \mathbb{R}^{r_1 \times \cdots \times r_d}, \mathbf{U}_k \in \mathbb{R}^{n_k \times r_k} \ \forall k \in [d]. \end{aligned} \tag{3.3}$$

Alternatively, the error-specified formulation of the Tucker approximation problem is given as

$$\begin{aligned} \min \quad & \prod_{j=1}^{d} r_j + \sum_{j=1}^{d} n_j r_j \\ \text{subject to } & \mathcal{G} \in \mathbf{R}^{r_1 \times \cdots r_d}, \mathbf{U}_k \in \mathbb{R}^{n_k \times r_k} \forall k \in [d] \\ \text{and} \quad & \|\mathcal{X} - (\mathcal{G} \times_1 \mathbf{U}_1 \cdots \times \mathbf{U}_d)\| \leq \epsilon \|\mathcal{X}\|. \end{aligned} \tag{3.4}$$

3.1.1 ST-HOSVD

We start with the state-of-the art algorithm that is capable of performing both rank-specified and error-specified formulations. Algorithm 5 showcases the d-way construction of the ST-HOSVD algorithm. This method approximately solves either (3.3) or (3.4) by unfolding the k^{th} mode of the input tensor, computing its left leading singular vectors (LLSV), and then performing a TTM with the result to truncate the k^{th} mode of rank r_k. Once all factor matrices have been computed, the truncated tensor has rank $\mathbf{r}$.

A relative error error of ϵ can be achieved by selecting r_k in the LLSV computation such that $\sum_{i=r_k+1}^{n_k} \sigma_i^2 \leq \epsilon^2 \|\mathcal{X}\|^2/d$, where σ_i is the i^{th} largest singular value of the k^{th}

unfolding, see [1] for more details. There are several algorithms one could choose in line 8 to compute $\mathbf{U}_k$. We assume that such computation is performed via the eigenvalue decomposition (EVD) of the Gram matrix $\mathbf{G}_{(k)}\mathbf{G}_{(k)}^{\intercal}$ as seen in algorithm 6.

Algorithm 5 ST-HOSVD

1 **Input:** Tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$
2 Ranks $\mathbf{r} = r_1, \ldots, r_d$ **OR** relative error tolerance $\epsilon > 0$
3 **Output:** TTensor $\mathcal{T}$ of ranks $\mathbf{r}$ with $\mathcal{T} \approx \mathcal{X}$ **OR** ERR $\equiv ||\mathcal{X} - \mathcal{T}|| \leq \epsilon ||\mathcal{X}||$
4 **function** ST-HOSVD($\mathcal{X}$, $\mathbf{r}$ or ϵ)
5 **if** ϵ is defined **then** $\bar{\epsilon} \leftarrow (\epsilon/\sqrt{d}) \cdot ||\mathcal{X}||$
6 $\mathcal{G} \leftarrow \mathcal{X}$
7 **for** $k = 1, \ldots, d$ **do**
8 $[U_k, \epsilon_k] \leftarrow \text{LLSV}(G_{(k)}, r_k \text{ or } \bar{\epsilon})$ ▷ r_k leading left sing. vectors of residual
9 $\mathcal{G} \leftarrow \mathcal{G} \times_k U_k^{\intercal}$ ▷ compress in mode k
10 **end for**
11 ERR $\leftarrow \sqrt{\sum_{k=1}^{d} \epsilon_k^2}$ ▷ equivalent to $||\mathcal{X} - \mathcal{T}||$
12 **return** $[\mathcal{G}, U_{1:d}, \text{ERR}]$ ▷ $\mathcal{T} \equiv \{\mathcal{G}; U_{1:d}\}$
13 **end function**

Algorithm 6 LLSV

1 **function** $\mathbf{U} = \text{LLSV}(\mathbf{Y}, r \text{ or } \epsilon)$
2 $\mathbf{S} = \mathbf{Y} \cdot \mathbf{Y}^{\intercal}$
3 $[\mathbf{U}, \Lambda] = \text{eig}(\mathbf{S})$
4 **return** $\mathbf{U}(:,\ 1:r)$
5 **end function**

3.1.2 Classic HOOI

The details of the HOOI algorithm are given in algorithm 7. HOOI is an alternative method for solving the rank-specified formulation of the Tucker approximation problem [15, 16, 17]. It is a block coordinate descent method and so it requires initial

factor matrices. Historically, the output factor matrices of ST-HOSVD have been used as input factor matrices for the HOOI algorithm, which is used simply to refine the approximation. However, random factor matrices can be used and generally no more than two iterations are required to get a good approximation. Often, only one iteration is enough to get a decent one.

HOOI iteratively updates each factor matrix by performing a TTM with all but the k^{th} factor matrix to obtain an intermediate tensor $\mathcal{Y}$. The k^{th}factor matrix is computed as the LLSV of $\mathbf{Y}_{(k)}$. The core tensor $\mathcal{G}$ can be computed once, at the end, or at the end of every iteration in order to computate a per-iteration approximation error. We introduce three optimizations for the HOOI algorithm in an attempt to make it more competitive against ST-HOSVD.

Algorithm 7 HOOI

Input: Tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$
Either Ranks $\mathbf{r} = r_1, \ldots, r_d$
Maximum Number of Iterations
Output: TTensor $\mathcal{T}$ of ranks $\mathbf{r}$ with $\mathcal{T} \approx \mathcal{X}$
function HOOI($\mathcal{X}$, $\mathbf{r}$ or ϵ)
  Initialize factor matrices $\mathbf{U}_{1:d}$ randomly
  $\mathcal{G} \leftarrow \mathcal{X}$
  for Maximum Number of Iterations **do**
    for $k = 1, \ldots, d$ **do**
      $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}_1^\intercal \times_2 \cdots \times_{k-1} \mathbf{U}_{k-1}^\intercal \times_{k+1} \mathbf{U}_{k+1}^\intercal \times_{k+2} \cdots \times_d \mathbf{U}_d^\intercal$
      $\mathbf{U}_k \leftarrow \text{LLSV}(\mathbf{Y}_{(k)}, r_k)$
    end for
  end for
  $\mathcal{G} \leftarrow \mathcal{Y} \times_d \mathbf{U}_d^\intercal$ ▷ update core
  return $[\mathcal{G}, U_{1:d}]$ ▷ $\mathcal{T} \equiv \{\mathcal{G}; U_{1:d}\}$
end function

3.1.3 HOOI's Dimension Trees Optimization

Adapting ranks in each HOOI iteration is a low order cost, however, the cost of TTMs is a factor of d more expensive than in ST-HOSVD. We can reduce the cost of TTMs by avoiding redundant computations. Notice that for $k = 1$ in algorithm 7 the following multi-TTM is computed $\mathcal{Y} = \mathcal{X} \times_2 \mathbf{U}_2^\intercal \times_3 \mathbf{U}_3^\intercal \cdots \times_d \mathbf{U}_d^\intercal$. At $k = 2$ the multi-TTM is $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}_1^\intercal \times_3 \mathbf{U}_3^\intercal \cdots \times_d \mathbf{U}_d^\intercal$. By comparing the two multi-TTMs we can see that $d-2$ TTMs are the same (namely 3 to d). So we can reuse results from one multi-TTM to the next by memoizing intermediate results. This idea, organized using so-called "dimension trees", was first used in the context of CP decompositions [20] and has been applied to Tucker computations as well [21, 22]. Section 3.1.3 shows an example dimension tree as we implement them for an order-6 tensor where each node represents the set of modes in which a TTM has not been performed. At the root of the tree, no TTMs have been performed, so the tensor is $\mathcal{X}$. Each notch in an edge of the tree represents a TTM in the labeled mode. At each leaf node, TTMs in all modes but one have been performed, so we update the factor matrix in that mode by performing LLSV. The core tensor $\mathcal{G}$ is updated at the last leaf node by perform a TTM between the (memoized) intermediate tensor and the factor matrix corresponding to the last leaf node. Algorithm 8 shows the HOOI iteration using dimension tree memoization implemented recursively.

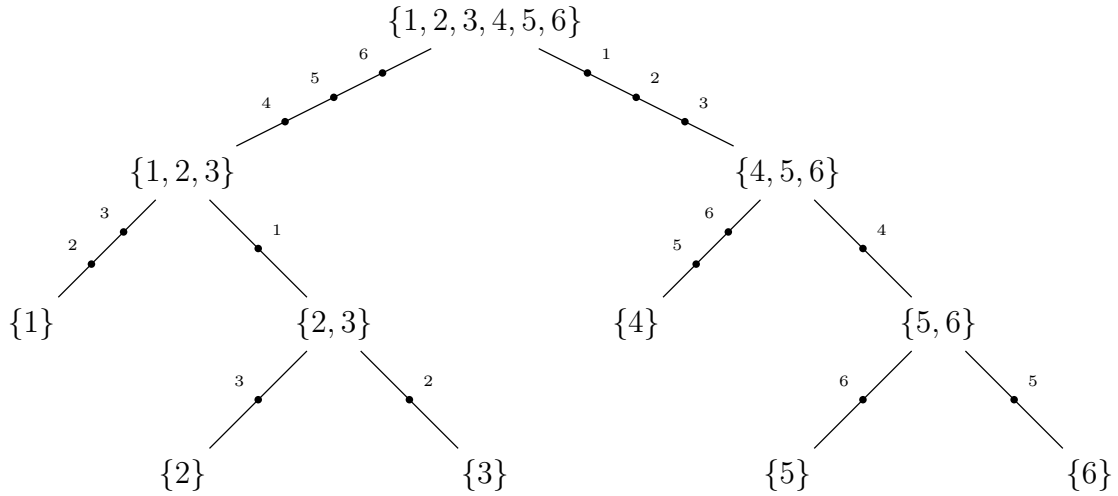


Figure 3.2: Illustration of multi-TTM memoization for an order-6 tensor. Each node in the tree shows the set of modes in which multiplication has not been performed. Each notch in an edge is a TTM in the labeled mode. Factor matrices are computed at each leaf node in the mode shown. $\mathcal{G}$ is updated in the last leaf node.

Algorithm 8 Recursive HOOI iteration via dimension trees

function $[\mathcal{G}, \{\mathbf{U}_k\}] =$HOOI-DT$(\mathcal{X}, \{\mathbf{U}_k\}, \mathbf{m}, \mathbf{r})$
  if length$(\mathbf{m}) = 1$ **then**
    $\mathbf{U}_m =$ LLSV$(\mathbf{X}_{(m)}, \mathbf{U}_m, r_m)$
    if $m = d$ **then**
      $\mathcal{G} = \mathcal{X} \times_d \mathbf{U}_m^\intercal$
    end if
  else
    Partition $\mathbf{m} = [\mu, \eta]$
    $\mathcal{X} = \mathcal{X} \bigtimes_{k \in \mu} \mathbf{U}_k^\intercal$
    $[\mathcal{G}, \{\mathbf{U}_k\}] =$ HOSI-DT$(\mathcal{X}, \{\mathbf{U}_k\}, \eta, \mathbf{r})$
    $\mathcal{X} = \mathcal{X} \bigtimes_{k \in \eta} \mathbf{U}_k^\intercal$
    $[\mathcal{G}, \{\mathbf{U}_k\}] =$ HOSI-DT$(\mathcal{X}, \{\mathbf{U}_k\}, \mu, \mathbf{r})$
  end if
end function

3.1.4 HOOI's Subspace Iteration Optimization

So far, we have assumed that the LLSVs of a matrix $\mathbf{A}$ are obtained as the eigenvectors of the Gram matrix, $\mathbf{A}\mathbf{A}^\intercal$. The next algorithmic improvement we introduce is to compute the leading left singular vectors by using subspace iteration. Algorithm 9 shows a single subspace iteration, but the computations could be repeated to improve accuracy.

Algorithm 9 LLSV via Subspace Iteration

```
1 function Q = LLSV(A, U, r)
2     G = U^T A
3     Z = A G^T
4     [Q, ~, ~] = QRCP(Z)
5 end function
```

We note that the input matrix $\mathbf{A}$ is $\mathbf{Y}_{(k)}$ from algorithm 7 or $\mathbf{X}_{(m)}$ from algorithm 8, which is the result of an all-but-one multi-TTM, and the input matrix $\mathbf{U}$ is the factor matrix from the previous HOOI iteration. This implies that the temporary matrix $\mathbf{G}$ in algorithm 9 is an unfolding of the core tensor corresponding to the current set of factor matrices. That is, the matrix multiplication in line 2 is a TTM, which we implement using existing TuckerMPI subroutines. The multiplication in line 3 is a tensor contraction in all modes but one between the core tensor and the result of an all-but-one multi-TTM, which is not implemented in TuckerMPI. Our parallel algorithm mimics the computation of the Gram matrix of a tensor unfolding, but it is a nonsymmetric operation and has different costs. Finally, we perform QR with col-

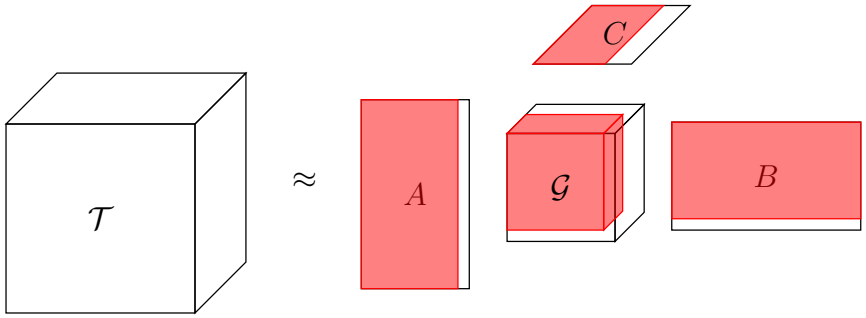


Figure 3.3: Adaptive HOOI

umn pivoting in line 4 to orthonormalize the subspace iteration result and also order the columns to aid in core analysis, which is discussed in section 3.1.5. We choose to do only a single subspace iteration because we use an accurate initialization (from the previous HOOI iteration) and because high accuracy of a HOOI subiteration is less of a priority than high accuracy of the full HOOI iteration.

3.1.5 HOOI's Adaptive Rank Optimization

A significant disadvantage of HOOI is that it solves only the rank-specified formulation of the Tucker approximation problem, whereas ST-HOSVD can adaptively select ranks based on a relative error tolerance. We propose a technique that allows HOOI to automatically adapt ranks to meet a user-specified relative error tolerance.

Recall that for the error-specified formulation, given an error tolerance ε and an initial rank estimate $\mathbf{r}$, our method adaptively finds a Tucker decomposition $\hat{\mathcal{X}} = [\mathcal{G}; \mathbf{U}_1, \ldots, \mathbf{U}_d]$ for a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ such that $\|\hat{\mathcal{X}} - \mathcal{X}\| \leq \varepsilon \|\mathcal{X}\|$. We start with a typical HOOI iteration using our initial rank estimate $\mathbf{r}$, partially compressing

our tensor in all modes except mode k and updating factor matrix $\mathbf{U}_k$ to be the first r_k left singular vectors of the partially compressed tensor. Once all modes have been processed in this manner, we check the error of the approximation at that point. Whereas in classical HOOI the core is only updated after the iterations, here we compute the core tensor at the end of every iteration and perform error analysis on it. To check the error, we use the identity that for orthonormal matrices $\mathbf{U}_1, \ldots, \mathbf{U}_d$ and $\mathcal{G} = \mathcal{X} \times_1 \mathbf{U}_1^\intercal \times \cdots \times_d \mathbf{U}_d^\intercal$, the approximation error can be written as $\|\mathcal{X} - \hat{\mathcal{X}}\|^2 = \|\mathcal{X} - \mathcal{G} \times_1 \mathbf{U}_1 \times \cdots \times_d \mathbf{U}_d\|^2 = \|\mathcal{X}\|^2 - \|\mathcal{G}\|^2$ ([1, Proposition 6.3]). If the current Tucker approximation is not sufficiently accurate, we increase all ranks by a factor α and perform the next HOOI iteration. If the current approximation satisfies the error threshold, then we can optimize over all rank truncations by analyzing the core tensor's entries. We can thus estimate the relative error in the approximation by computing $\|\mathcal{G}\|$, and choosing the next rank $\mathbf{r}$ so that $\|\mathcal{G}(\mathbf{1} : \mathbf{r})\|^2 \approx (1 - \varepsilon^2)\|\mathcal{X}\|^2$. Specifically, we solve the optimization problem

$$\begin{aligned} \min_{\mathbf{r}} \quad & \prod_{j=1}^{d} r_j + \sum_{j=1}^{d} n_j r_j, \\ \text{subject to } & \|\mathcal{G}(\mathbf{1} : \mathbf{r})\|^2 \geq (1 - \varepsilon^2)\|\mathcal{X}\|^2. \end{aligned} \tag{3.5}$$

This computes the leading subtensor of $\mathcal{G}$ that minimizes the size of the Tucker approximation and also satisfies the error threshold. Note that any subtensor of the core, along with the corresponding columns of the factor matrices, is a valid

Tucker approximation with error determined by the norm of the core subtensor. The optimal subtensor need not be a leading one, but we order factor matrix columns to concentrate the weight of $\mathcal{G}$ towards the entry of smallest index value so that the heuristic of searching over only leading subtensors is reasonable.

If such a rank $\mathbf{r}$ exists, we set our next rank as the solution to (3.5) and truncate to that rank before iterating. If no $\mathbf{r}$ exists, our current rank is too small, so we increase it by a some factor α before the next iteration. Typically, $\alpha \approx 2$ is sufficient. The details of this algorithm, are described in algorithm 10. In practice, we do the optimization problem above in a way that minimizes the memory footprint. We compute the cumulative sum of the squared core and then consider all values of this cumulative sum squared tensor to solve the optimization problem (3.5).

Algorithm 10 Adaptive HOOI

function PERFORMCOREANALYSIS($\mathcal{G}, \epsilon, \mathbf{r}$)
  if $||\mathcal{G}||^2 \geq (1-\epsilon^2)||\mathcal{X}||^2$ **then**
    Find $\mathbf{r} = \arg\min ||\mathcal{G}(1:\mathbf{r})||^2$
      subject to $||\mathcal{G}(1:\mathbf{r})||^2 \geq (1-\epsilon^2)||\mathcal{X}||^2$

    Truncate $\mathcal{G}, A, B, C$ according to $\mathbf{r}$
  else
    $\mathbf{r} = \alpha\ \mathbf{r}$
    Increase columns of A, B, C according to $\mathbf{r}$
  end if
  return r
end function

3.2 The TuckerMPI Library

TuckerMPI uses P processors organized into a d-dimensional $P_1 \times \cdots \times P_d$ grid such that $P = \prod_{i=1}^{d} P_i$ and that each processor stores a $1/P$ fraction of $\mathcal{X}$. Our analysis will assume $\mathcal{X} \in \mathbb{R}^{n \times \cdots \times n}$ and $\mathcal{G} \in \mathbb{R}^{r \times \cdots \times r}$ to simplify cost comparison across algorithms.

3.2.1 TuckerMPI's ST-HOSVD

ST-HOSVD's Computational Complexity

The cost of LLSV in line 8 is given by

$$\sum_{j=1}^{d} \left(\frac{r^{j-1} n^{d-j+2}}{P} + \mathcal{O}(n^3) \right) \approx \frac{n^{d+1}}{P} + \mathcal{O}(dn^3),$$

where the first term is the cost of computing the $n \times n$ Gram matrix and the second term is the cost of sequentially computing the EVDs to leading order. After $\mathbf{U}_k$ is computed, $\mathcal{Y}$ is truncated by performing the TTM in line 9, which costs

$$2 \sum_{j=1}^{d} \frac{r^j n^{d-j+1}}{P} \approx 2 \frac{r n^d}{P}.$$

Computing the Gram matrix is a factor of $n/2r$ more expensive than the TTM and is the dominant cost for $n \gg r$. Sequentially truncating $\mathcal{Y}$ leads to decreasing dimensions, so the algorithm is typically dominated by the first Gram matrix computation. Note that the EVD is not parallelized, which can be a barrier to parallel scaling when

a single tensor dimension is large. We summarize the leading order ST-HOSVD flops cost in table 3.1 (shown in red).

ST-HOSVD's Communication Complexity

TuckerMPI's parallel algorithm for LLSV explicitly forms the Gram matrix, $\mathbf{G} = \mathbf{Y}_{(k)}\mathbf{Y}_{(k)}^{\mathsf{T}}$, where $\mathbf{Y}_{(k)}$ is redistributed (if necessary) to a 1D column layout across P processors, and then sequentially computes the EVD of $\mathbf{G}$. After redistribution of $\mathbf{G}$, each processor computes a local Gram matrix which can be sum-reduced (or all-reduced) prior to the EVD. At iteration k, the number of entries in $\mathcal{Y}$ is $r^{j-1}n^{d-j+1}$. The Gram matrix that is computed in each mode is of size $n \times n$, so the total communication cost is dn^2 for the all-reduce. Thus, the communication cost is given by

$$\sum_{j=1}^{d} \left(\frac{r^{j-1}n^{d-j+1}}{P} \cdot \frac{P_j - 1}{P_j} + \mathcal{O}(n^2) \right) \approx \frac{n^d}{P} \cdot \frac{P_1 - 1}{P_1} + O(dn^2),$$

where we assume the redistribution cost is dominated by the first mode. However, note that there is no redistribution cost in mode j if $P_j = 1$. Finally, the parallel TTM also requires communication to perform a sum-reduce of local TTM results. Since the output of the TTM is largest in the first mode (of size rn^{d-1}/P), the communication cost of TTMs to leading order cost is

$$\sum_{j=1}^{d} \frac{r^j n^{d-j}}{P}(P_j - 1) \approx \frac{rn^{d-1}}{P}(P_1 - 1).$$

Again, note there is no communication cost in mode j if $P_j = 1$. Because the largest data communicated occurs in mode 1, processor grids with $P_1 = 1$ are typically the fastest for ST-HOSVD (as we observe in our experiments). We summarize the ST-HOSVD communication costs in table 3.2 (shown in red).

3.2.2 TuckerMPI's HOOI

HOOI's Computational Complexity

Since HOOI is an iterative algorithm for Tucker decomposition, we analyze the cost of one HOOI iteration. Each HOOI iteration requires d multi-TTMs, in all modes but mode-j, and d LLSV computations to update factors matrices, in all modes. Once the factor matrices have been updated, the core tensor $\mathcal{G}$ is obtained by performing a TTM with the last factor matrix $\mathbf{U}_d$. The cost of computing d multi-TTMs is given by

$$2d \sum_{i=1}^{d} \frac{r^i n^{d-i+1}}{P} \approx 2d \frac{r n^d}{P}.$$

The cost of each TTM decreases, so the first term in the summation (i.e. the first TTM) dominates. Multiplying the cost of the first TTM by d yields the cost of d multi-TTMs (i.e. one HOOI iteration). The cost of computing LLSV is given by

$$d \frac{r^{d-1} n^2}{P} + \mathcal{O}(dn^3),$$

where the first term is the cost of computing the Gram matrix $\mathbf{Y}_{(k)}\mathbf{Y}_{(k)}^{\mathsf{T}}$ and the second term is the cost of computing the EVD. Finally, the core tensor at the end of each HOOI iteration is obtained by performing a TTM in mode-d with the intermediate tensor $\mathcal{Y}$ and $\mathbf{U}_d$, which has a cost of $2 \cdot {}^{nr^d}/_{P}$ and is a lower order term. We summarize the leading order cost per HOOI iteration as implemented by TuckerMPI in table 3.1 (shown in red).

HOOI's Communication Complexity

The communication cost of each iteration of HOOI is dominated by multi-TTMs and LLSV computations. Each TTM in the multi-TTM requires communication to perform a sum reduction to form $\mathcal{Y}$. Communication is required along the processor dimension corresponding to the mode in which a TTM is performed. The size of $\mathcal{Y}$ decreases with each TTM, so the communication cost of a multi-TTM is dominated by the first TTM. Each HOOI iteration performs d multi-TTMs, where one iteration updates the factor matrix in the first mode. The cost of communication for the multi-TTMs is given by

$$\sum_{j=1}^{d}\left(\sum_{i=1}^{j-1}\frac{r^i n^{d-i+2}}{P}(P_i-1)+\sum_{i=j+1}^{d}\frac{r^{i-1}n^{d-1+1}}{P}(P_i-1)\right)$$
$$\approx (d-1)\frac{rn^{d-1}}{P}(P_1-1)+\frac{rn^{d-1}}{P}(P_2-1).$$

The first term corresponds to the $d-1$ TTMs performed in the 1st mode and the second term corresponds to TTMs performed in the 2nd mode (for the multi-TTM in all but the 1st mode).

Communication is also required when computing the LLSV in each mode. Using the same LLSV algorithm as in ST-HOSVD, the Gram matrix is computed in parallel followed by a sequential EVD. Computing the Gram matrix requires an all-to-all to redistribute $\mathbf{Y}_{(k)}$ so that it is stored in 1D-column layout. After redistribution $\mathbf{Y}_k\mathbf{Y}_k^\intercal$ is computed in parallel by performing local matrix-matrix multiplications that are sum-reduced to obtain the Gram matrix. The cost of communication for the LLSV is given by

$$\frac{r^{d-1}n}{P}\sum_{i=1}^{d}\left(\frac{P_i-1}{P_i}\right)+dn^2,$$

where the first term is the cost of all-to-all communication and the second term is the cost of sum reduction of the Gram matrix for one HOOI iteration (i.e. d calls to LLSV). We summarize the HOOI communication costs as implemented by TuckerMPI in Table 3.2 (shown in red).

3.2.3 TuckerMPI's Dimension Tree

Dimension Tree Computational Complexity

The flops cost of performing multi-TTMs using dimension-trees is given by

$$4\sum_{i=1}^{d/2}\frac{r^i n^{d-i+1}}{P} + \mathcal{O}\left(d\sum_{i=d/2+1}^{d} r^i n^{d-i+1}\right) \approx 4\frac{rn^d}{P},$$

where the first term is the cost of computing the TTMs in the first two branches (left and right of the root) in the dimension tree and the second term is the cost of computing the TTMs in all remaining branches. The largest TTMs in the first two branches dominate, so the cost of multi-TTMs is $4 \cdot {}^{rn^d}/_{P}$ (i.e. the first TTM in each branch), which is a factor of ${}^{d}/_{2}$ improvement over computing multi-TTMs directly. This cost is summarized in Table 3.1.

Dimension Tree Communication Complexity

Since the first TTM in each of the two multi-TTMs off the root dominate, the communication cost of multi-TTMs is given by

$$\sum_{i=1}^{d/2}\frac{r^i n^{d-i-1}}{P}\left(P_i - 1 + P_{d-i+1} - 1\right) \approx \frac{rn^{d-1}}{P}\left(P_1 + P_d - 2\right).$$

When traversing the right branch in the dimension tree shown in section 3.1.3, TTMs are performed in the first ${}^{d}/_{2}$ modes starting with mode 1. The communication cost associated with TTMs in the right branch is the cost of a reduce-scatter on local

data of size $rn^{d-1}/P \cdot (P_1 - 1)$, which yields the first term. The second term is due to the communication cost associated with traversing the left branch in section 3.1.3. TTMs in the left branch are performed in the last $d/2$ modes starting with mode d. We perform left branch TTMs in reverse order because the mode d TTM achieves higher local TTM performance due to the layout of the local tensor in memory. The communication cost associated with TTMs in the left branch is the same as the first term, except that the reduce-scatter is performed in the P_d processor grid dimension. Therefore, processor grids with $P_1 = P_d = 1$ are typically the fastest for HOOI algorithms employing the dimension tree optimization (as we observe in our experiments).

As shown in tables 3.1 and 3.2, introducing dimension-trees memoization reduces the flops cost of TTMs in HOOI by a factor of $d/2$ and the communication cost by a factor of $d - 1$ in the first term.

3.2.4 TuckerMPI's Subspace Iterations

Subspace Iteration Computational Complexity

Each subspace iteration requires two matrix-matrix multiplications and one QR decomposition. The first matrix-multiplication corresponds to the TTM $\mathcal{G} = \mathcal{Y} \times_k \mathbf{U}_{(k)}^\intercal$ (in the notation of algorithm 7) and the second computes the tensor contraction $\mathbf{Y}_{(k)}\mathbf{G}_{(k)}^\intercal$. The total computational cost of performing the TTM and contraction in

each HOOI iteration is $4d \cdot {}^{nr^d}/_{P}$. The cost of the QR decomposition of the matrix $\mathbf{Z} \in \mathbb{R}^{n \times r}$ in each HOOI iteration is $\mathcal{O}(dnr^2)$, where we assume a sequential QR decomposition. The total cost of performing subspace iteration in each mode across an entire HOOI iteration is given by

$$4d\frac{nr^d}{P} + \mathcal{O}(dnr^2).$$

As shown in table 3.1, the cost of LLSV using subspace iteration is a factor of ${}^{1}/_{4} \cdot {}^{n}/_{r}$ cheaper than the cost of LLSV via the Gram matrix. When comparing the sequential EVD to the sequential QR decomposition, the cost of the latter is a factor of $\mathcal{O}\left(\left({}^{n}/_{r}\right)^2\right)$ faster.

Subspace Iteration Communication Complexity

Subspace iteration requires communication in the TTM, tensor contraction, and QR decomposition in each mode. The communication cost of the TTM is given by ${}^{r^d}/_{P} \cdot (P_k - 1)$, where P_k corresponds to the number of processors in the k^{th} mode. The tensor contraction requires redistribution of both tensors via all-to-all communication steps. However, the all-to-all cost is a lower order term since it is a factor of P_k cheaper than the communication cost associated with the TTM. Once the contraction is performed, a sum reduction followed by a broadcast is required to ensure that all processors can independently compute local QR decompositions. The communication cost of the QR decomposition is given by $2nr$ since $\mathbf{Z} \in \mathbf{R}^{n \times r}$ and must be communicated twice. As

shown in table 3.2, the total communication cost of the LLSV calls within an iteration of HOOI using subspace iteration is given by

$$\frac{r^d}{P}\sum_{j=1}^{d}(P_j - 1) + 2dnr.$$

3.2.5 TuckerMPI's Adaptive Rank

Core Analysis Computational Complexity

The cost of one RA-HOOI iteration is the same as one iteration of HOOI given in table 3.1, but with the possible additional cost of performing analysis on the core tensor $\mathcal{G}$ to adapt the ranks for the next iteration. We solve the optimization problem given in (3.5) exhaustively by computing the norm and corresponding size of every leading subtensor. This can be done using only $\mathcal{O}(dr^d)$ operations by employing a multidimensional prefix sum computation across the squares of the core entries. Because computational cost tends to be dominated by the rest of the HOOI iteration, we perform the core analysis sequentially, though the prefix sums are readily parallelizable.

Assuming that this analysis is performed sequentially, the cost of the core analysis is $\mathcal{O}(r^d)$. The cost of the core analysis is dominated by the cost of computing a cumulative sum of entries in $\mathcal{G}$ and finding the smallest entry which meets the relative error tolerance. Performing these operations requires $\mathcal{O}(r^d)$ flops. Since we need n/r to be

Algorithm	LLSV		TTM		Core Analysis
HOOI iteration	**Gram + Eig**	$d\frac{n^2 r^{d-1}}{P} + \mathcal{O}(dn^3)$	**Direct**	$2d\frac{rn^d}{P}$	$\mathcal{O}(dr^d)$
	Sub. Iter.	$4d\frac{nr^d}{P} + \mathcal{O}(dnr^2)$	**Dim. Tree**	$4\frac{rn^d}{P}$	
ST-HOSVD	$\frac{n^{d+1}}{P} + \mathcal{O}(dn^3)$		$2\frac{rn^d}{P}$		-
RA-HOSI-DT	$\ell\left(4d\frac{nr^d}{P} + \mathcal{O}(dnr^2)\right)$		$\ell\left(4\frac{rn^d}{P}\right)$		$\ell\left(\mathcal{O}(dr^d)\right)$

Table 3.1: Leading order flops costs of LLSV (Gram + Eig and Subspace Iteration), multi-TTM (Direct and dimension-trees) and Core Analysis algorithmic choices for HOOI and a comparison between ST-HOSVD and HOOI with Subspace Iteration and dimension-trees (HOSI-DT) optimizations. We assume ℓ iterations of HOSI-DT are performed

large for HOOI to improve performance over ST-HOSVD, the cost of sequential core analysis can be performed in parallel, but we expect that the cost of communication would outweigh the benefits of parallelizing this operation.

Core Analysis Communication Complexity

At the end of a HOOI iteration, $\mathcal{G}$ is distributed across all processors, so it must be gathered on a single processor in order to perform analysis. Since the entire core tensor must be communicated, the all-gather cost is r^d per HOOI iteration. We demonstrate in section 3.3 that the sequential cost of core analysis is typically negligible.

3.3 Results

This section presents a comparison of the running time (strong scaling and running time breakdown) and compression (error vs. time and error vs. compression ratio)

Algorithm	LLSV		TTM		Core Analysis
HOOI iteration	Gram + Eig	$\frac{nr^{d-1}}{P}\sum_{i=1}^{d}\frac{P_i-1}{P_i}+dn^2$	Direct	$(d-1)\frac{rn^{d-1}}{P}(P_1-1)+\frac{rn^{d-1}}{P}(P_2-1)$	r^d
	Sub. Iter.	$\frac{r^d}{P}\sum_{i=1}^{d}(P_i-1)+2dnr$	Dim. Tree.	$\frac{rn^{d-1}}{P}(P_1-1)+\frac{rn^{d-1}}{P}(P_d-1)$	
ST-HOSVD	$\frac{n^d}{P}\frac{P_1-1}{P_1}+dn^2$		$\frac{rn^{d-1}}{P}(P_1-1)$		-
RA-HOSI-DT	$t\left(\frac{r^d}{P}\sum_{i=1}^{d}(P_i-1)+2dnr\right)$		$t\left(\frac{rn^{d-1}}{P}(P_1+P_d-2)\right)$		$\ell\left(r^d\right)$

Table 3.2: Leading order bandwidth costs of LLSV (Gram + Eig and Subspace Iteration), multi-TTM (Direct and dimension-trees) and Core Analysis algorithmic choices for HOOI. For reference, we include a comparison between ST-HOSVD and HOOI with Subspace Iteration and dimension-trees (HOSI-DT). We assume a processor grid of $P = (P_1 \times \cdots \times P_d)$ and that ℓ iterations of HOSI-DT are performed.

performance of the various Tucker algorithms presented in this work. All algorithms were implemented using the TuckerMPI (C++/OpenMPI) library [9].

Computing platform. Our experiments were conducted on NERSC Perlmutter (CPU partition). The system consists of 3072 compute nodes with dual-socket AMD EPYC 7763 64-core CPUs. Each socket has 4 Non-Uniform Memory Access (NUMA) regions for a total of 8 NUMA regions per node. Each NUMA region has 64 GB of DRAM memory, therefore each CPU socket has 256 GB of DRAM for, a total of 512 GB of memory per node.

Experiments. We perform experiments on synthetic tensors that are randomly generated and tensors obtained from real applications. We use 3-way and 4-way tensors for the synthetic experiments, and three real datasets: Miranda [23] (3-way), HCCI [24] (4-way), and SP [25] (5-way). The real datasets are described in more

detail in section 3.3.2. Experiments performed on synthetic tensors are performed in single precision, while experiments on real datasets are performed in single or double precision depending on their storage precision on disk. Strong scaling experiments are performed on the synthetic tensors. We show running time breakdown of both real and synthetic experiments. For synthetic tensors we show the running time breakdown at small and large scale to highlight how each step in a given algorithm scales. For real tensors we vary the error tolerance and starting ranks to show how performance breakdowns vary. Compression performance experiments are performed only on the real datasets.

Even for a fixed number of processors P, the d-way processor grid has a significant effect on all algorithms. As described in section 3.2.1, STHOSVD benefits from processor grids with $P_1 = 1$, and HOOI variants using dimension trees are theoretically more efficient when $P_1 = P_d = 1$. In addition, for modes with small tensor dimension, a large processor dimension in that mode may cause load imbalance due to uneven division. In all experiments, we test all algorithms on a variety of grids, including those we expect to benefit individual algorithms, and we report the fastest observed running times.

3.3.1 Strong Scaling on Synthetic Tensors

First, we present strong scaling experiments on the 3-way and 4-way synthetic tensors to demonstrate the parallel scaling of HOOI, HOOI-DT, HOSI, HOSI-DT, and STHOSVD. We choose tensor dimensions to maximize the size of the tensor that can fit on a single node (in single precision).

For synthetic input, we generate tensors by forming a Tucker-format tensor of specified rank and adding a specified level of noise. Thus, these experiments are performed for the rank-specified formulation of the Tucker approximation problem to recover the input. We run for two iterations for each variant of HOOI even though we often have a sufficiently accurate approximation after a single iteration. We include overhead due to core analysis for the error-specified formulation in the experiments on the real datasets. The largest 3-way tensor that fits into single-node memory is a tensor of size $3750 \times 3750 \times 3750$. We generate this tensor to have a rank of 30 in all modes. Similarly, we construct the 4-way tensor of size $560 \times 560 \times 560 \times 560$ with Tucker ranks $(10, 10, 10, 10)$.

Figures 3.4 and 3.5 shows the strong scaling results of the HOOI variants and STHOSVD on up to 4096 cores for the 3-way and 4-way synthetic datasets. We observe that STHOSVD scales well to 64 cores, attaining a speedup of 15.2× over the single core STHOSVD run. STHOSVD continues to scale up to 2048 cores, but achieves only a modest speedup of 1.3× over the 64 core run. This is due to

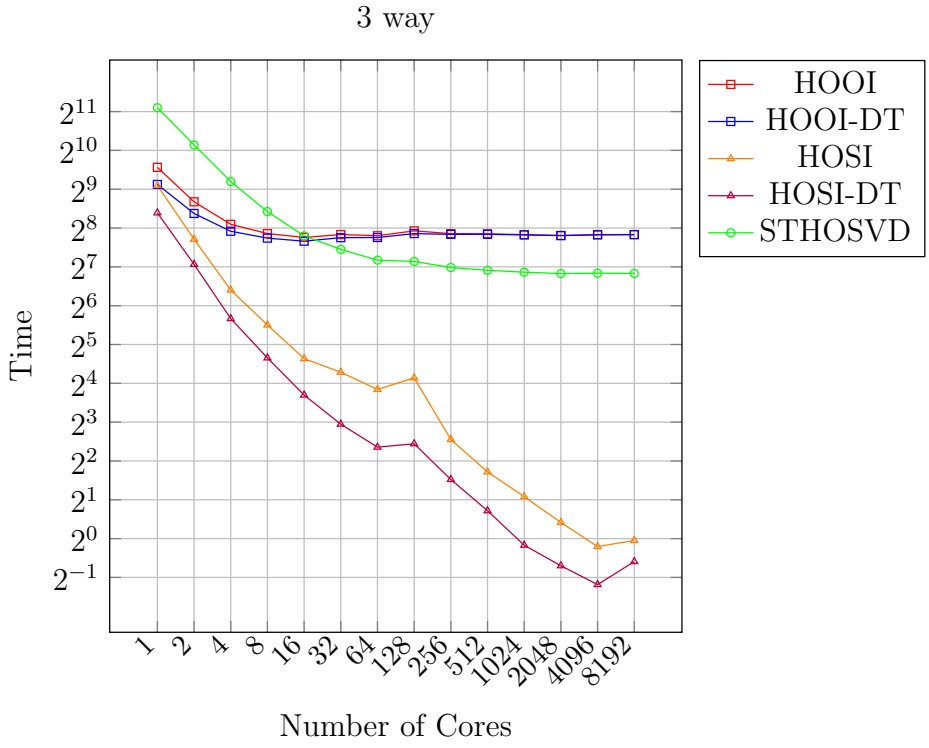


Figure 3.4: Strong scaling comparison of Tucker algorithms in single precision using a 3-way $3750 \times 3750 \times 3750$ input tensor

TuckerMPI's limitation of having a sequential EVD implementation. In contrast, the 4-way STHOSVD strong scaling experiment shows good scaling up to 8192 cores, achieving a speedup of 937× over the single core run. This difference in STHOSVD performance is explained by the tensor dimension: a sequential EVD of a matrix of dimension 560 does not become the bottleneck until P is large.

When comparing the two HOOI variants (which use Gram SVD), we observe that HOOI-DT yields a sequential speedup of 1.4× over HOOI's direct TTM implementation for the 3-way tensor. For the 4-way tensor, HOOI-DT achieves a sequential speedup of 5.4× faster than HOOI. When comparing parallel scaling in the 3-way

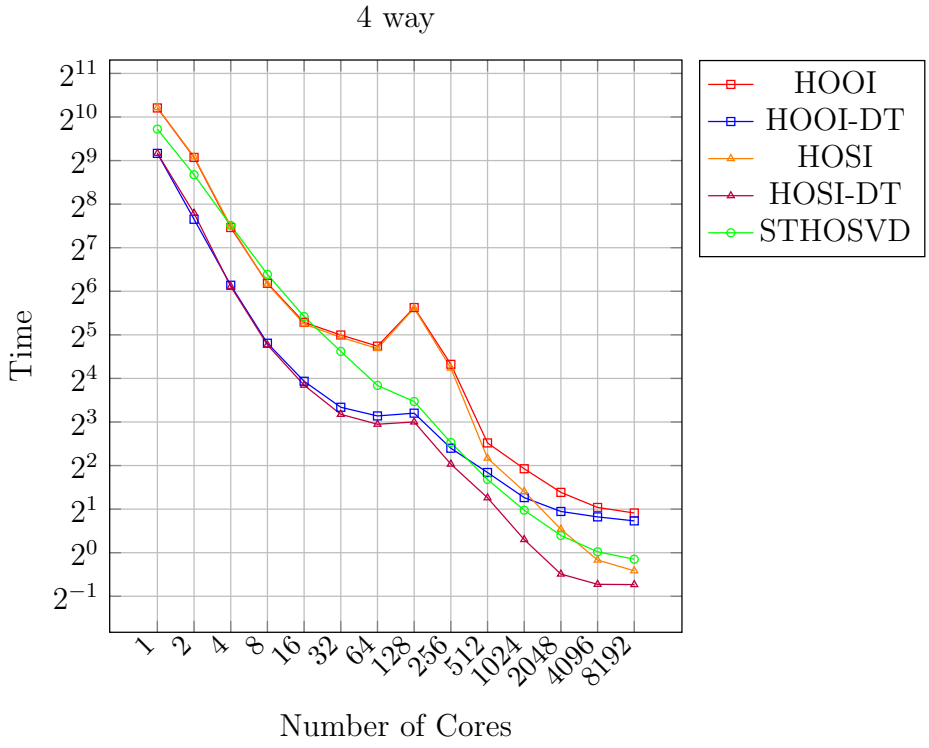


Figure 3.5: Strong scaling comparison of Tucker algorithms in single precision using a 4-way $560 \times 560 \times 560 \times 560$ input tensor

case, we see that HOOI and HOOI-DT scale to 16 cores with a speedup of 3.5× and 2.8×, respectively, over their single core runs. However, neither variant scales beyond 16 cores for the 3-way tensor because of the sequential EVD bottlenecks. For the 4-way tensor, HOOI and HOOI-DT scale to 8192 cores with a speedup of 629× and 346×, respectively, over their single core runs. The performance of HOOI and HOOI-DT degrades at 128 cores (single node) because both variants are memory-bandwidth bound, and we saturate bandwidth at 64 cores. HOOI and HOOI-DT continue scaling beyond 128 cores (multi-node scaling) because memory bandwidth increases. As can be seen in the 4096 core plots of figures 3.6 and 3.7, HOOI and HOOI-DT suffer

from the problem of the sequential EVD, and they are approximately twice as slow as STHOSVD because they do twice as many EVDs over two iterations.

HOSI and HOSI-DT show significantly better scaling on the 3-way tensor when compared to STHOSVD and the HOOI variants because of the difference in LLSV subroutines. HOSI-DT achieves sequential speedups of 6.5× and 1.7× over STHOSVD and HOOI-DT, respectively. The HOSI variants scale to 4096 cores with HOSI-DT achieving significant parallel speedups of 259× and 515× over STHOSVD and HOOI-DT, respectively. HOSI-DT is also the fastest Tucker variant for the 4-way experiment attaining speedups of 1.5× and 2.9× over STHOSVD and HOOI-DT, respectively when comparing the best running times of each algorithm. HOSI and HOSI-DT exhibit similar memory bandwidth scaling behavior as the HOOI variants where performance degrades at 128 cores (single node) and continues to scale beyond 128 cores (multi-node scaling). These can be seen on figures 3.6 and 3.7. We chose to showcase the breakdown using 1 core and using 4096 cores.

3-way. Observing the single-node scaling (1 to 128) of the 3-way experiment, we notice that all HOOI algorithms outperform STHOSVD, with HOSI and HOSI-DT being much further ahead of the competition since the large $\frac{n}{r}$ ratio of this experiment implies a LLSV bottleneck and these two algorithms avoid that. Here, STHOSVD is much slower because it does a more expensive LLSVs before the TTMs whereas the HOOI algorithms reduce this cost by performing the TTMs beforehand. Starting

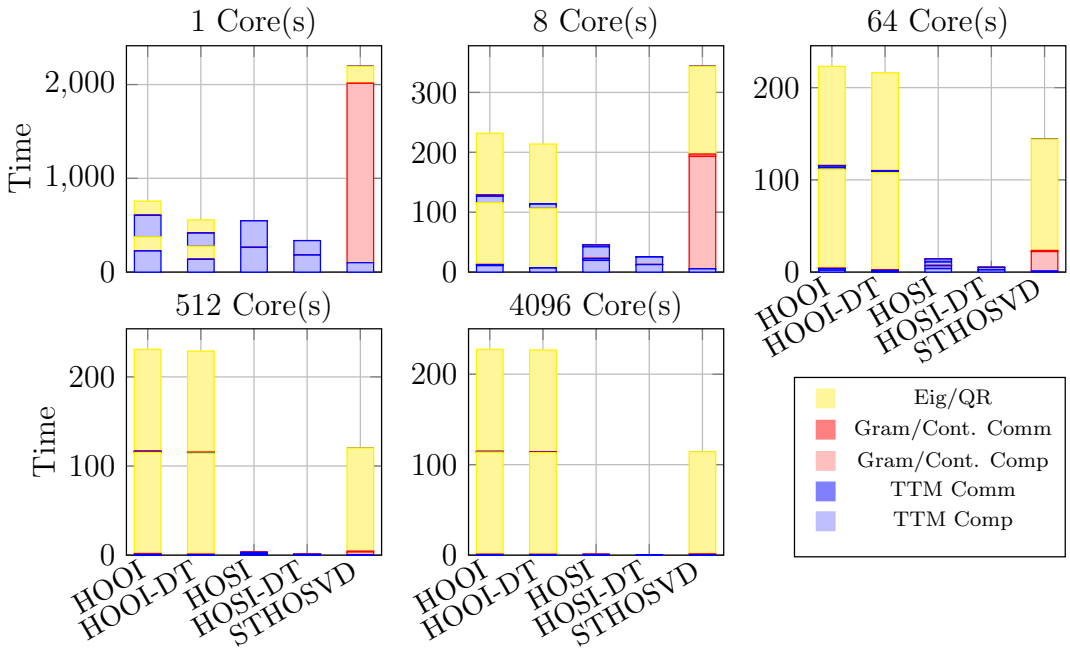


Figure 3.6: Running time breakdown for the synthetic 3-way tensor

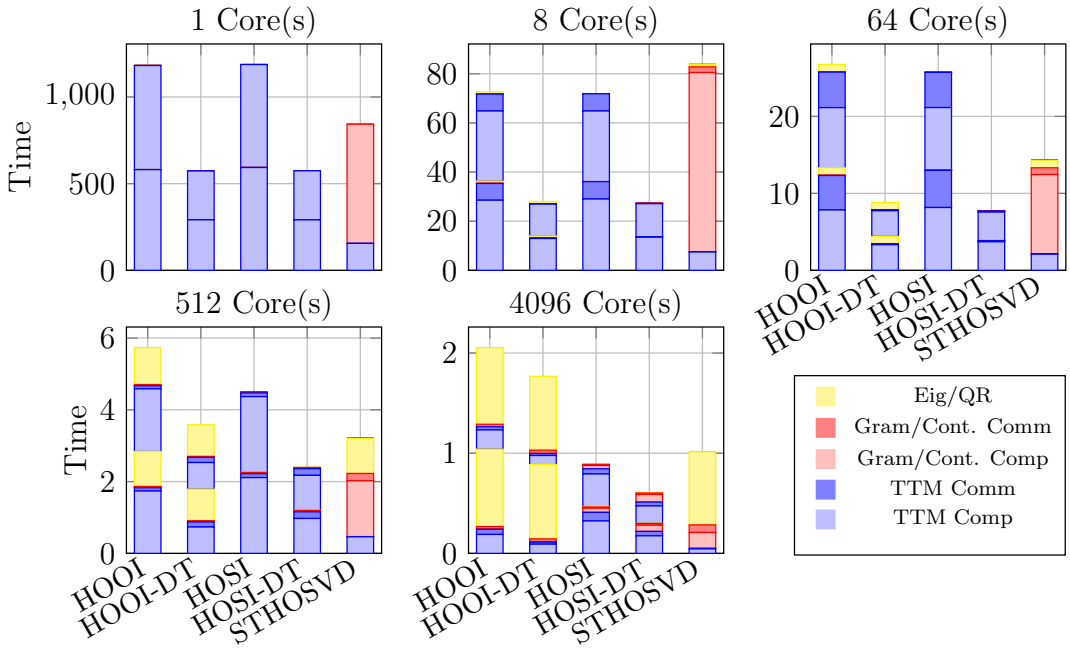


Figure 3.7: Running time breakdown for the synthetic 4-way tensor

from 32 nodes, STHOSVD begins to outperform HOOI and HOOI-DT due to the cost of the LLSV. Figure 3.6 demonstrates that the cost of LLSV is the same for HOOI, HOOI-DT and STHOSVD, but because HOOI must perform two iterations, it takes twice as long. In fact, we see that the three algorithms stagnate at large scaling due to TuckerMPI's limitation of having a sequential eigenvalue computation. Though HOSI and HOSI-DT still perform two iterations, neither of them need to pay the cost of the sequential eigenvalue decomposition. Even with a sequential QR decomposition, they still scale best, with the lesser TTM cost of HOSI-DT making it the best out of these five algorithms.

For the 3-way synthetic tensor on 1 core, it can be seen that the cost of the Gram computation is the bottleneck for STHOSVD. The TTM cost for the dimension tree algorithms are cheaper, as expected. HOSI-DT is faster than HOOI-DT simply because of the smaller cost for the LLSV computations. The two iterations for all HOOI algorithms are faster than the 'single iteration' for STHOSVD. However, that is not the case for the 4096 cores experiment. Now, the TTM costs for all algorithms are negligible due to the parallel scaling. The Gram computation for STHOSVD is also negligible now for the same reasons. The bottleneck at this scale is now the Eigenvalue computation. The reason why HOOI, HOOI-DT, and STHOSVD stagnate over the high number of cores on Figure 3.4 is because of TuckerMPI's limitation of having a sequential eigenvalue computation, and the reason why HOOI and HOOI-DT are

twice as expensive on multi-node experiments is because they must do two iterations, this fact can be visualized on the breakdown for the 4096 cores experiment.

4-way. In this experiment, HOOI-DT and HOSI-DT are the ones that get a comparative headstart since the small $\frac{n}{r}$ ratio of this experiment implies a TTM bottleneck and these two algorithms avoid that. They diverge around 128 cores for the same reasons mentioned above as HOOI-DT must pay the cost of two expensive eigenvalue computations. For that matter, STHOSVD still eventually catches up to HOOI-DT, but now this happens at 512 cores as opposed to 32. For this reason, HOSI eventually closes the gap to HOSI-DT, with STHOSVD being not too far behind simply because of the smaller $\frac{n}{r}$.

3.3.2 Performance on Simulation Datasets

We turn our focus for the error-specified comparison of our best algorithm, HOSI-DT, and the state-of-the-art, STHOSVD. The data sets are decomposed using three error tolerances; 0.1 ("high compression"), 0.05 ("mid compression"), and 0.01 ("low compression"). Furthermore, we showcase HOSI-DT through three different types of starting ranks for each error tolerance. Perfect starting ranks are the same as the final ranks of STHOSVD given the maximum relative error threshold. We overshoot and undershoot the same starting ranks by 25% above and below to force our algorithm to respectively increase and decrease ranks on the first iteration. We cap the number

of iterations for HOSI-DT at 3. Though all three iterations are shown in the Error vs Time and Error vs Size plots, the running time breakdown plots show the breakdown only for however many iterations it took for HOSI-DT to reach the desired error threshold. For example, the top right plot of figure 3.11 it can be seen that the HOSI-DT (Over) 0.1 threshold reached the desired at the first iteration, so we don't show the breakdown for the second iteration despite its total time being shown on figure 3.10.

Miranda (3-way)

The Miranda dataset is a three-dimensional simulation data of the density ratios of non-reacting flow of viscous fluids [23]. Each of its dimensions is 3072, and it is stored in single precision requiring 115 GB. Our experiments use 1024 cores (8 nodes) for all algorithms.

Figure 3.8 demonstrates that for all error tolerances, three iterations of HOSI-DT combined is faster than STHOSVD. But as mentioned earlier, we focus on the least amount of iterations required to reach the desired error threshold. It is in high- and mid-compression where we find the most speedup. Precisely, perfect ranks achieve speedups of 82× for high-compression and 25× for mid-compression, undershooting the ranks achieves speedups of 91× for high-compression and 35× for mid-compression, and overshooting the ranks achieves speedups of 156× for high-

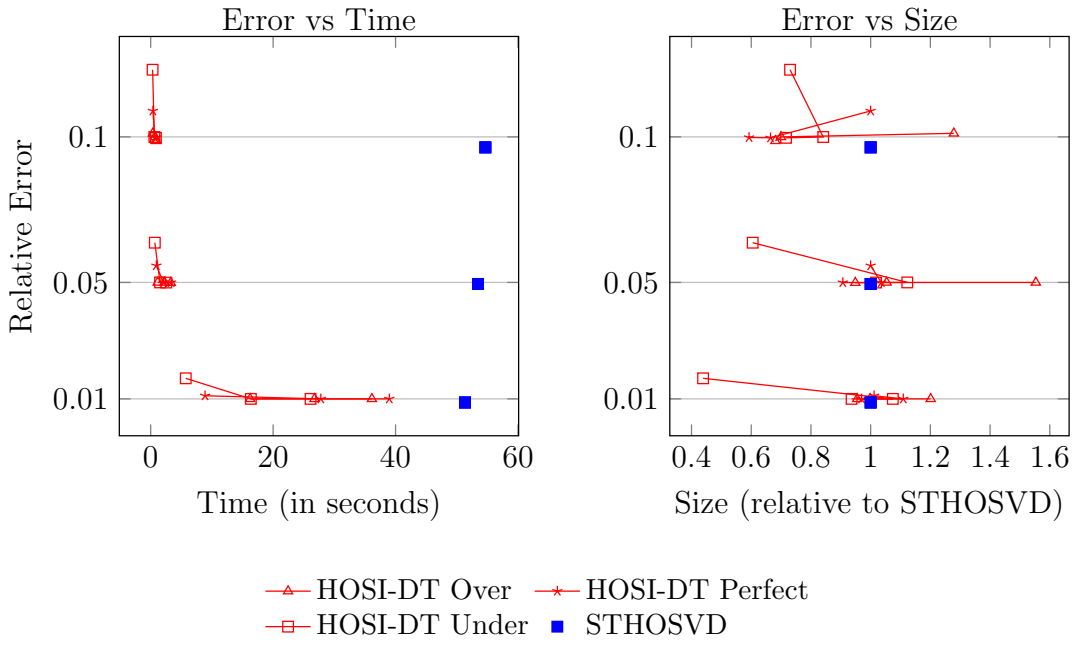


Figure 3.8: Progression of time, error, and relative size over 3 iterations of rank-adaptive HOSI-DT on the Miranda dataset using 1024 cores.

compression and 47× for mid-compression. Low-compression is the first scenario where we observe nonnegligible costs of the core analysis subroutine. For high-compression, the best relative compression ratio is 69% which occurs at perfect ranks, mid-compression achieves a 10% improvement using perfect ranks, and low-compression has better compression at 6% when underestimating the ranks.

HCCI (4-way) and SP (5-way)

We combine the discussion of the HCCI and SP datasets results, as the results are qualitatively similar. The Homogeneous Charge Compression Ignition (HHCI) dataset is generated from a numerical simulation of combustion [24]. The dimension

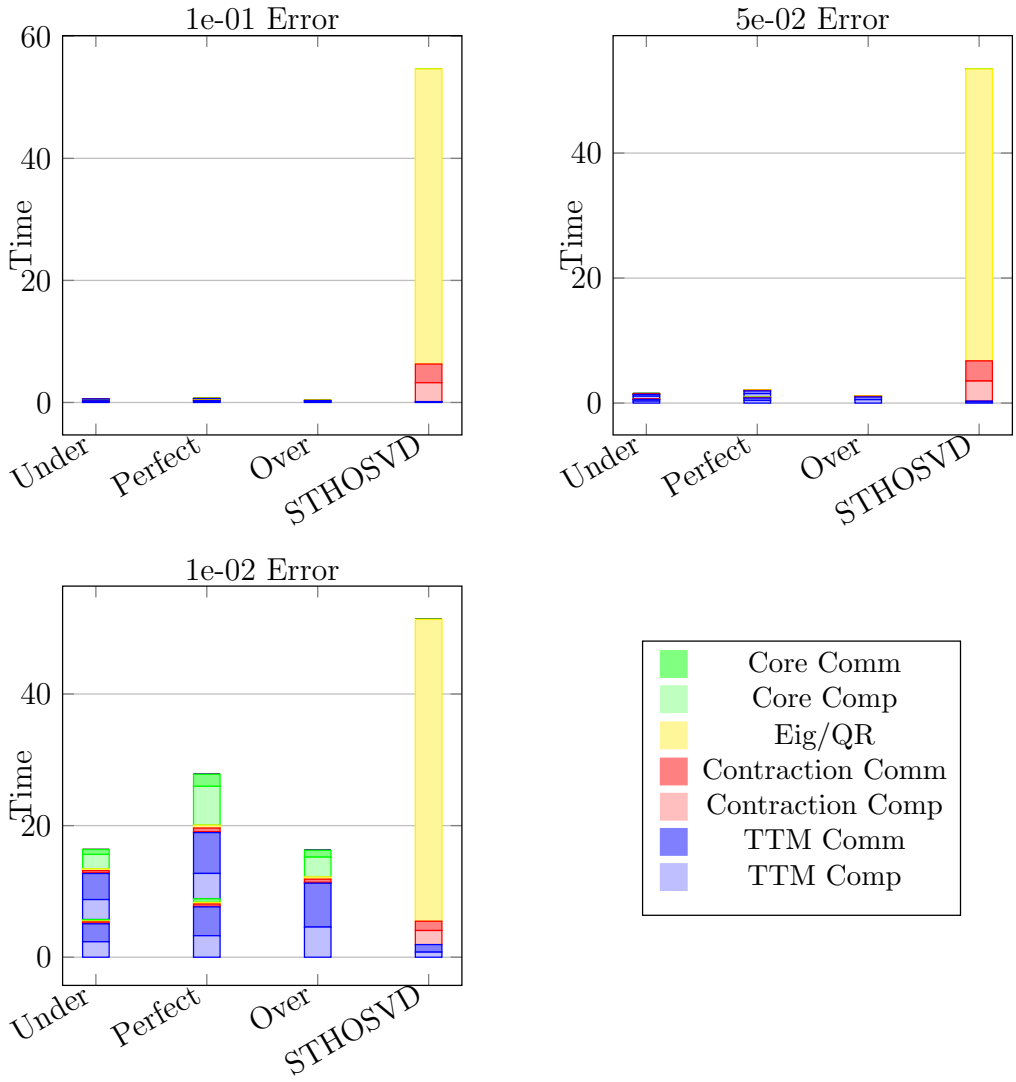


Figure 3.9: Running time breakdown for the Miranda dataset using 1024 cores under different levels of compression.

of the 4-way dataset is $672 \times 672 \times 33 \times 626$ stored in double precision for a total of 75 GB. Thus, we can fit it on a single node and use all 128 cores. The first two modes are spatial dimensions, the third mode corresponds to 33 variable, and the fourth mode corresponds to time steps. The SP dataset is generated from the simulation of a statistically stationary planar methane-air flame [25]. This 5-way dataset has dimensions $500 \times 500 \times 500 \times 11 \times 400$ stored in double precision and requires 4.4 TB in storage. For these experiments, we use 2048 cores (16 nodes). The first three modes are spatial dimensions, the fourth mode corresponds to 11 variables, and the last mode corresponds to time steps.

In the case where we are dominated by the TTMs, the comparisons between HOSI-DT and STHOSVD are less extreme. Figure 3.10 shows that on low-compression, STHOSVD is faster than any of the starting ranks of HOSI-DT to get to the desired threshold. However, for high- and mid-compression HOSI-DT achieves speedups when overshooting the ranks, specifically 1.9× for high-compression and 1.4× for low-compression, neither of which achieved better compression. Figure 3.11 shows the breakdown times of these speedups. However, HODI-DT achieves better compression with perfect and under ranks for all error tolerances, but always requiring three iterations to do so.

Figure 3.12 shows that we can typically obtain better compression after three iterations. For example, overestimating the ranks for low compression yields a speedup

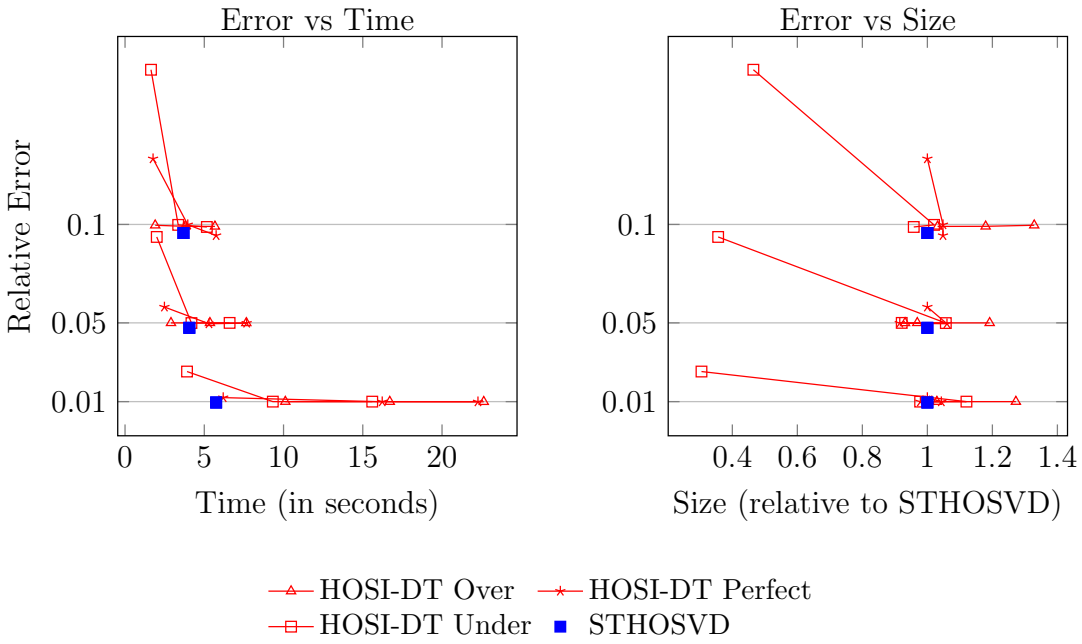


Figure 3.10: Progression of time, error, and relative size over 3 iterations of rank-adaptive HOSI-DT on the HCCI dataset using 128 cores.

of 1.1× after 1 iteration, but we do not obtain better compression. Similar to HCCI, three iterations produces a smaller Tucker approximation but takes over twice as long. However, for high compression, starting from perfect and underestimates of the ranks achieve a 27% and 8% improvement on compression over STHOSVD after two iterations, respectively. In another example, figure 3.13 shows that when starting from perfect estimates of the ranks for mid compression, HOSI-DT gets the desired error tolerance and same compression ratio in less time than STHOSVD, with HOSI-DT achieving a 1.4× speedup.

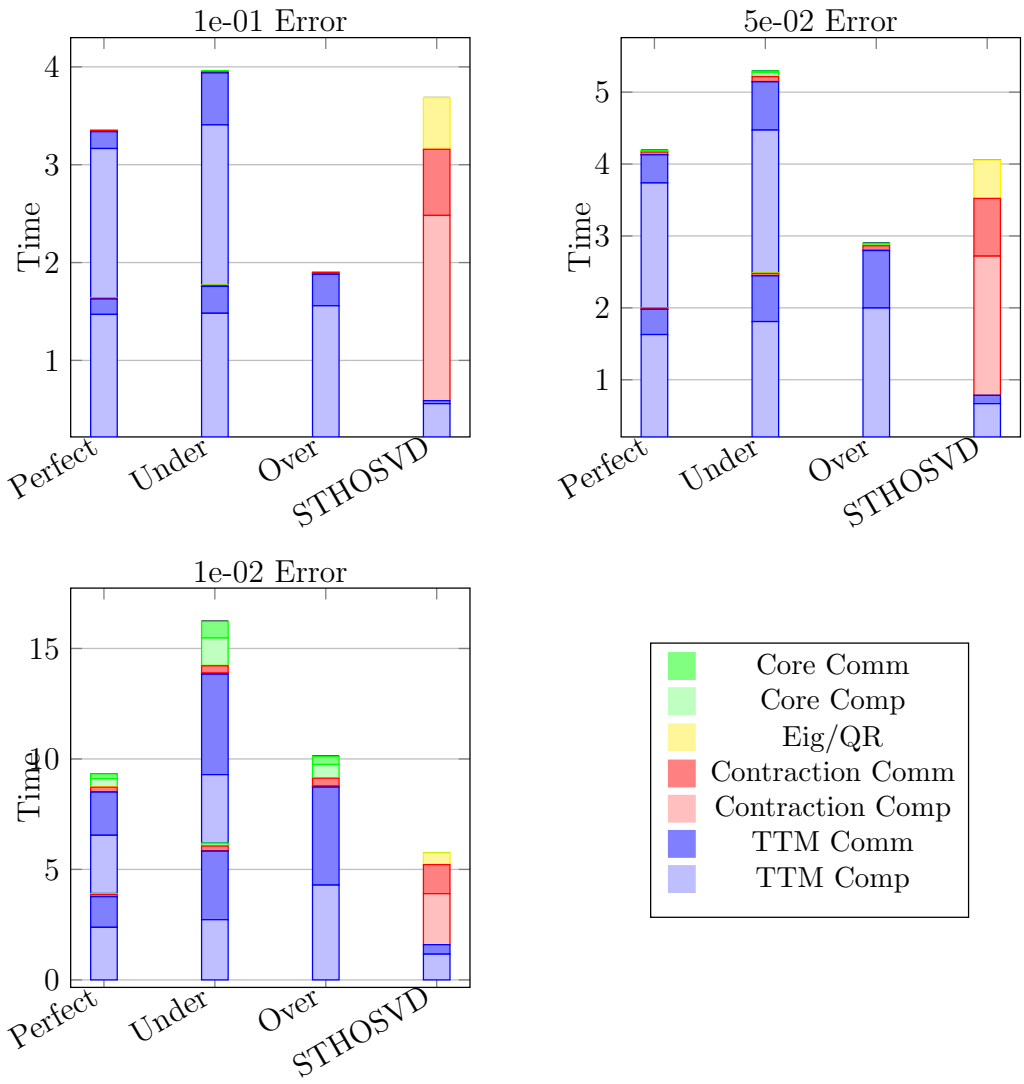


Figure 3.11: Running time breakdown for the HCCI dataset using 128 cores under different levels of compression.

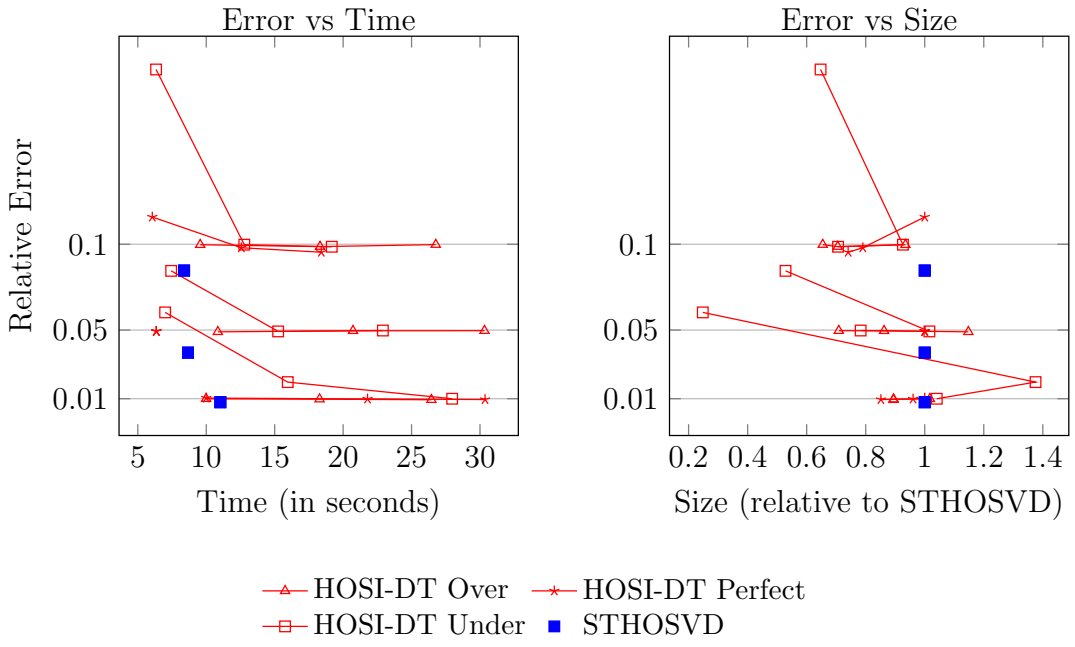


Figure 3.12: Progression of time, error, and relative size over 3 iterations of rank-adaptive HOSI-DT on the SP dataset using 2048 cores.

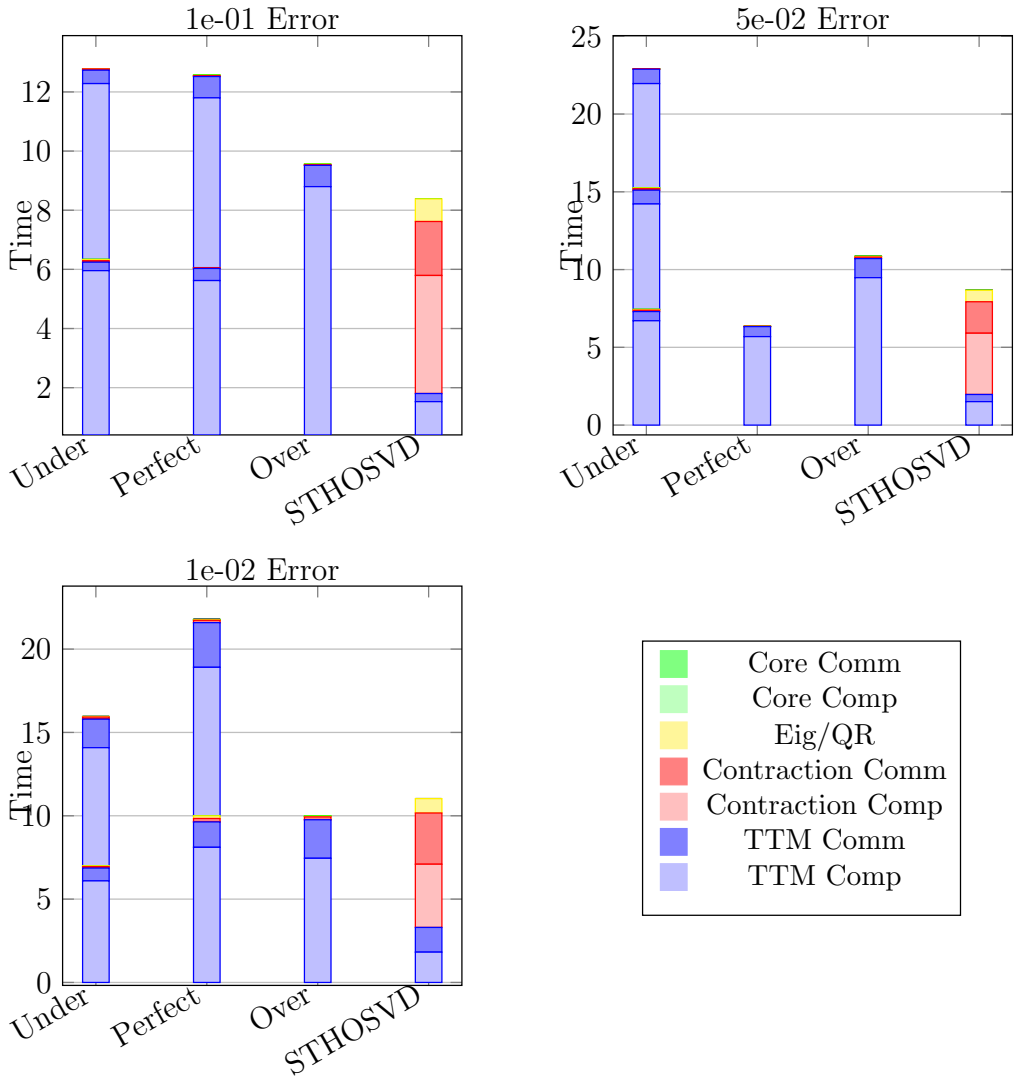


Figure 3.13: Running time breakdown for the SP dataset using 2048 cores under different levels of compression.

CONCLUSION

Chapter 2 explores the realm of fast matrix multiplication algorithms. These algorithms improve on the computational complexity of the classical algorithm, but the problem of the complexity of matrix multiplication remains open. Many different techniques are used to search for algorithms that attempt to shorten the gap between known upper bounds and proven lower bound. The techniques explored in this chapter search for algorithms that are cyclic invariant, which decrease the number of search parameters in a numerical optimization algorithm. With the introduction of CI_CP_DGN, we have been able to obtain many valid algorithms with this property for multiple base case dimensions. Our on-going work attempts to search for further symmetries in the algorithms we have found while searching for cyclic invariant algorithms. Once a more robust comprehension of these symmetries is achieved, our goal is to study how these symmetries lay the foundation for more structure to be exploited in the search for fast matrix mutliplication algorithms. We intend to submit this work for publication in the near future. The code associated with this project can be found in **https://github.com/Jv7Pinheiro/FastMatrixMultiplyAlgorithmsSearch.git**.

Chapter 3 describes three optimization for the HOOI algorithm to make it a competitive alternative to the state-of-the art algorithm ST-HOSVD. The dimension-tree, subspace iteration optimizations reduce the computational complexity of each HOOI iteration, and the rank adaptive optimization generalizes the method to solve the error-specified Tucker approximation problem.

Based on the complexity analysis and experimental results, we conclude that our parallel RA-HOSI-DT computes Tucker approximations of comparable error in less time than TuckerMPI's implementation of STHOSVD in two important scenarios: (1) when large individual tensor dimensions create sequential EVD bottlenecks, and (2) when individual ratios between input tensor and core tensor dimensions are large. In the first case, because of the scalability of RA-HOSI-DT, we observe very large speedups with large P. In the second case, our theoretical analysis suggests a speedup roughly proportional to n/r. However, we observe that while the number of flops is reduced compared to STHOSVD, the local matrix computation performance degrades because the smallest matrix dimension in the computation becomes r instead of n. That is, if the ranks are very small, then local matrix computations with RA-HOSI-DT run far below peak processor performance and are instead limited by the memory bandwidth. This memory bandwidth bottleneck is the reason RA-HOSI-DT loses scalability when using all cores on a single node and is the main reason the theoretical computational cost analysis doesn't match empirical performance at scale.

RA-HOSI-DT requires an input estimate of the final core ranks. While priori knowledge is not required, we observe that slight overestimates of the final ranks yield sufficiently accurate solutions often in the first iteration. When ranks are underestimated, HOOI must iterate until an overestimate is discovered, after which a single iteration yields convergence.

Furthermore, in solving the error-specified optimization problem, we highlight that RA-HOSI-DT often identifies Tucker approximations with better compression ratios than STHOSVD. This is due in large part to the flexibility afforded by the RA-HOSI-DT core analysis step to shift ranks across modes to maximize overall compression, as opposed to STHOSVD, which makes greedy decisions at each mode. If compression ratio is more important than time, taking more HOOI iterations can help to improve accuracy and often reduce ranks further.

All of our parallel implementations have been carried through in the TuckerMPI library. We intend to merge our work with the main branch soon, which can be found **https://gitlab.com/tensors/TuckerMPI.git**. This work has been submitted for publication.

Bibliography

[1] Grey Ballard and Tamara G. Kolda. *Tensor Decompositions for Data Science*. Cambridge University Press, 2025.

[2] Rodney Johnson and Aileen McLoughlin. "Noncommutative Bilinear Algorithms for 3 x 3 Matrix Multiplication". In: *SIAM Journal on Computing* 15.2 (1986), pp. 595–603. DOI: 10.1137/0215043.

[3] Grey Ballard et al. "The geometry of rank decompositions of matrix multiplication II: 3x3 matrices". In: *arXiv* 1801.00843 (2019).

[4] Kathryn Rouse. *On the Efficiency of Algorithms for Tensor Decompositions and Their Applications*. Available from Dissertations and Theses at Wake Forest University; ProQuest Dissertations and Theses Global. (2051225085). 2018. URL: https://wake.idm.oclc.org/login?url=https://www.proquest.com/dissertations-theses/on-efficiency-algorithms-tensor-decompositions/docview/2051225085/se-2.

[5] Jan R. Magnus and H. Neudecker. "The Commutation Matrix: Some Properties and Applications". In: *The Annals of Statistics* 7.2 (1979), pp. 381–394. DOI: 10.1214/aos/1176344621.

[6] Woody Austin, Grey Ballard, and Tamara G. Kolda. "Parallel Tensor Compression for Large-Scale Scientific Data". In: *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium*. May 2016, pp. 912–922.

DOI: 10.1109/IPDPS.2016.67. URL: https://www.computer.org/csdl/proceedings/ipdps/2016/2140/00/2140a912-abs.html.

[7] V. T. Chakaravarthy et al. "On Optimizing Distributed Tucker Decomposition for Dense Tensors". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2017, pp. 1038–1047. DOI: 10.1109/IPDPS.2017.86.

[8] Jee Choi, Xing Liu, and Venkatesan Chakaravarthy. "High-performance Dense Tucker Decomposition on GPU Clusters". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. Dallas, Texas: IEEE Press, 2018, 42:1–42:11. URL: http://dl.acm.org/citation.cfm?id=3291656.3291712.

[9] Grey Ballard, Alicia Klinvex, and Tamara G. Kolda. "TuckerMPI: A Parallel C++/MPI Software Package for Large-Scale Data Compression via the Tucker Tensor Decomposition". In: *ACM Transactions on Mathematical Software* 46.2 (June 2020). ISSN: 0098-3500. DOI: 10.1145/3378445. URL: https://dl.acm.org/doi/10.1145/3378445.

[10] Rafael Ballester-Ripoll, Peter Lindstrom, and Renato Pajarola. "TTHRESH: Tensor Compression for Multidimensional Visual Data". In: *IEEE Transactions on Visualization and Computer Graphics* 26.9 (2020), pp. 2891–2903. DOI: 10.1109/TVCG.2019.2904063.

[11] Wouter Baert and Nick Vannieuwenhoven. "Algorithm 1036: ATC, An Advanced Tucker Compression Library for Multidimensional Data". In: *ACM Transactions on Mathematical Software* 49.2 (June 2023), pp. 1–25. DOI: 10.1145/3585514.

[12] Saibal De et al. "Hybrid Parallel Tucker Decomposition of Streaming Data". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '24. Zurich, Switzerland: Association for Computing Machinery, 2024.

ISBN: 9798400706394. DOI: 10.1145/3659914.3659934. URL: https://doi.org/10.1145/3659914.3659934.

[13] Nick Vannieuwenhoven, Raf Vandebril, and Karl Meerbergen. "A New Truncation Strategy for the Higher-Order Singular Value Decomposition". In: *SIAM Journal on Scientific Computing* 34.2 (2012), A1027–A1052. DOI: 10.1137/110836067. eprint: http://dx.doi.org/10.1137/110836067. URL: http://dx.doi.org/10.1137/110836067.

[14] Wolfgang Hackbusch. *Tensor Spaces and Numerical Tensor Calculus*. 2nd. Springer International Publishing, 2019. ISBN: 9783030355548. DOI: 10.1007/978-3-030-35554-8.

[15] Pieter M Kroonenberg and Jan De Leeuw. "Principal component analysis of three-mode data by means of alternating least squares algorithms". In: *Psychometrika* 45 (1980), pp. 69–97. DOI: 10.1007/BF02293599.

[16] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. "On the best rank-1 and rank-$(r_1, r_2, ..., r_n)$ approximation of higher-order tensors". In: *SIAM journal on Matrix Analysis and Applications* 21.4 (2000), pp. 1324–1342. DOI: 10.1137/S089547989834699.

[17] Arie Kapteyn, Heinz Neudecker, and Tom Wansbeek. "An approach to n-mode components analysis". In: *Psychometrika* 51 (1986), pp. 269–275. DOI: 10.1007/BF02293984.

[18] Venkatesan T Chakaravarthy et al. "On optimizing distributed Tucker decomposition for dense tensors". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 1038–1047. DOI: 10.1109/IPDPS.2017.86.

[19] Linjian Ma and Edgar Solomonik. "Accelerating alternating least squares for tensor decomposition by pairwise perturbation". In: *Numerical Linear Algebra with Applications* e2431 (2022), pp. 1–33. DOI: 10.1002/nla.2431.

[20] Anh-Huy Phan, Petr Tichavsky, and Andrzej Cichocki. "Fast Alternating LS Algorithms for High Order CANDECOMP/PARAFAC Tensor Factorizations". In: *IEEE Transactions on Signal Processing* 61.19 (Oct. 2013), pp. 4834–4846. ISSN: 1053-587X. DOI: 10.1109/TSP.2013.2269903.

[21] Oguz Kaya and Yves Robert. "Computing dense tensor decompositions with optimal dimension trees". In: *Algorithmica* 81 (2019), pp. 2092–2121. DOI: 10.1007/s00453-018-0525-3.

[22] Rachel Minster, Zitong Li, and Grey Ballard. "Parallel Randomized Tucker Decomposition Algorithms". In: *SIAM Journal on Scientific Computing* 46.2 (2024), A1186–A1213. DOI: 10.1137/22m1540363. URL: https://doi.org/10.1137/22M1540363.

[23] Kai Zhao et al. "SDRBench: Scientific Data Reduction Benchmark for Lossy Compressors". In: *IEEE International Conference on Big Data*. 2020, pp. 2716–2724. DOI: 10.1109/BigData50022.2020.9378449.

[24] Ankit Bhagatwala, Jacqueline H. Chen, and Tianfeng Lu. "Direct numerical simulations of HCCI/SACI with ethanol". In: *Combustion and Flame* 161.7 (2014), pp. 1826–1841. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2013.12.027. URL: https://www.sciencedirect.com/science/article/pii/S0010218014000030.

[25] Hemanth Kolla et al. "Velocity and Reactive Scalar Dissipation Spectra in Turbulent Premixed Flames". In: *Combustion Science and Technology* 188.9 (2016), pp. 1424–1439. DOI: 10.1080/00102202.2016.1197211. eprint: https://doi.org/10.1080/00102202.2016.1197211. URL: https://doi.org/10.1080/00102202.2016.1197211.

CURRICULUM VITAE

João Pinheiro

deolj19@wfu.edu | jv7pinheiro.github.io

EDUCATION

Wake Forest University, Winston-Salem, North Carolina

Masters of Science in Computer Science, GPA 3.9, May 2025

Wake Forest University, Winston-Salem, North Carolina

Bachelor of Science in Applied Mathematics, May 2023, Cum Laude

Minors in Computer Sciences and in Schools, Education, and Society (SES)

Westhill Institute, Mexico City

IB Diploma, May 2019

RESEARCH EXPERIENCE

In Computer Science

- Tensor Decompositions, Dr. Grey Ballard & Dr. Aditya Devarakonda, Summer 2023 - Spring 2025

- Fast Matrix Multiplication, Dr. Grey Ballard & Dr. Frank Moore, Summer 2023 - Spring 2025

- Machine Learning & Computer Vision, Dr. Paul Pauca, Fall 2022 - Spring 2023

In Education

- Latin American Education, Dr. Betina Wilkinson, Fall 2019 - Spring 2020
- Educational Computer Science, Dr. Ali Sakkal, Spring 2023, (SES Minor Senior Project)

PROFESSIONAL EXPERIENCE

Research Assistant, *Department of Computer Science WFU*, July 2023 - Spring 2025

Teaching Assistant, *Department of Computer Science WFU*, Spring 2025

Academic Tutor, *Math and Stats Center WFU*, January 2021 - Spring 2025

Student Tutor, *Latinx Mentoring Initiative at Latino Community Services*, August 2019 - December 2020

Mentor, *Big Brother Big Sisters*, January 2019 - December 2020

PUBLICATIONS AND PRESENTATIONS

- K. Cui, Z. Shao, G. Larsen, V.P. Pauca, S. Alqahtani, D. Segurado, J. Pinheiro, M. Wang, D. Lutz, R. Plemmons, and M. Silman. 2024. PalmProbNet: A Probabilistic Approach to Understanding Palm Distributions in Ecuadorian Tropical Forest via Transfer Learning. Proceedings of the 2024 ACM Southeast Conference (ACMSE '24). ACM, 272-277. https://doi.org/10.1145/3603287.3651220

- Searching for Cyclic-Invariant Fast Matrix Multiplication Algorithms. Presented at Workshop on Sparse Tensor Computations in October 2023 and Graduate School of Arts & Sciences Graduate Student and PostDoc Research Day in March 2024 (Both were poster presentations of the same project).

- Introducing a Parallel Implementation For HOOI Tucker Tensor Decomposition With Rank Adaptivity. Presented at the SIAM Conference on Computational Science and Engineering in March 2025.

- João Pinheiro, Grey Ballard, Frank Moore, Pratyush Mishra, The geometry of rank decompositions of matrix multiplication III: 4x4 matrices (in preparation)

- João Pinheiro, Grey Ballard, Aditya Devarakonda, Introducing Parallel Rank-Adaptive Higher Order Orthogonal Iteration. (submitted to SC25)

SPECIAL SKILLS

Programing Languages: MATLAB, C/C++, OpenMP/OpenMPI, Java, Python, Git, Slurm, Bash, Latex/Tikz Fluent in English, Spanish, Portuguese, and Italian.