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

Dedico este trabalho aos meus pais, que sob muito sol, fizeram-me chegar até aqui, na sombra.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF ABREVIATIONS

CP Decompositions

CP: Cynical Polynomial

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

STHOSVD: 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 in its own chapter. The first project applies tensor decompositions on searches for new algorithms for fast matrix multiplication. The second project focuses on the development of scalable, adaptive methods for compressing massive data sets. Here, the abstracts for the two projects separately, in chronological order in which I began working on them.

Searching For Fast Matrix Multiply Algorithms using Cyclic-Invariant 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 M 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, and cyclic invariant decompositions are known to exist for 3x3 and 4x4 matrix multiplication as well. Cyclic invariance 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 these solutions, which involve one third of the variables of generic solutions, using the damped Gauss-Newton optimization method along with heuristic rounding techniques, and we summarize the algorithms discovered so far.

Parallel Higher-Order Orthogonal Iteration for Tucker Decomposition with Rank Adaptivity: Higher Order Orthogonal Iteration (HOOI) is an algorithm that uses block coordinate descent optimization to compress an input tensor into a Tucker format approximation. Classical HOOI requires specifying the ranks, or dimensions of the core tensor, and seeks to minimize the approximation error. We introduce HOOI-Adapt, a novel error-specified algorithm that adaptively selects the ranks of the Tucker tensor in order to maximize the compression subject to the

error threshold. We implement HOOI-Adapt using the distributed-memory parallel TuckerMPI library and employ memoization to reduce the computational cost. Furthermore, we show that when the ranks are small, HOOI-Adapt has lower computational cost compared to the alternative Sequentially Truncated Higher-Order SVD (ST-HOSVD) algorithm. Our benchmarks demonstrate that our parallel implementation of HOOI-Adapt outperforms TuckerMPI's ST-HOSVD for small ranks when scaled to terabyte-sized input datasets on NERSC's Perlmutter platform.

Introduction

1.1 Tensor and Their Subparts

1.1.1 What Is A Tensor?

There is no widely agreed definition of a tensor; different fields define it differently. In March 2024, Mathematics Professor Thomas Lam at NYU conducted an online informal survey on mathematical conventions [1]. The results of one of its questions can be seen on Figure 1.1. Though the target audience for this survey were mostly mathematicians and mathematics enthusiats, we can still observe a lerge variaty in definitions of a tensor. But in the great words of my advisor *"84% of people are wrong"*, and so we reach the definition: **a tensor is a multidimensional array**.

In other words, a **tensor** is a d-way array, where d is referred to as the order of the tensor. Before we move on any further, a bit of notation is required. 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

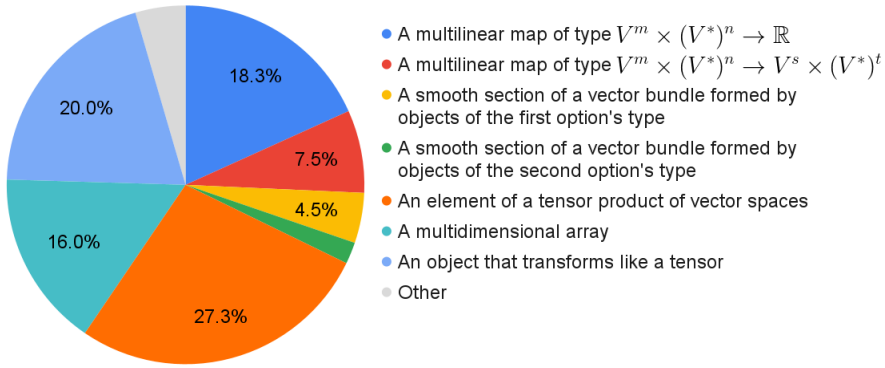
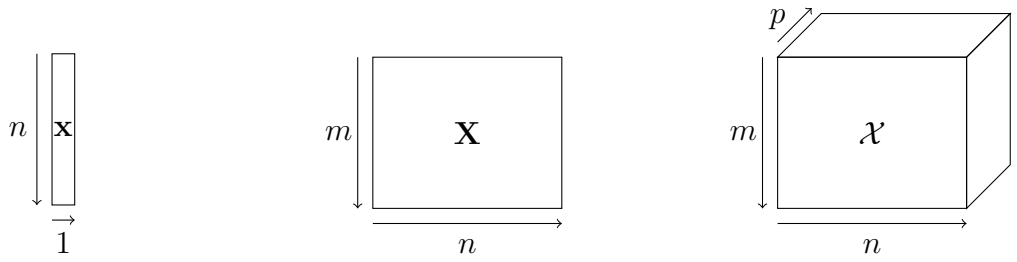


Figure 1.1: What Is A Tensor?

$i_1, \cdots, i_d$). For the sake of simplification, let $[n] \equiv [1, \cdots, n]$, and furthermore let $[m] \otimes [n] \equiv \{(i,j)|i \in [m], j \in [n]\}$. Some lower order tensors have other names:

- A **scalar** is a 'zero-dimensional' tensor. This is simply a number
- A **vector** is a one-dimensional array of scalars that represent a collection of measurements. This can be visualzied on Figure 1.2a. 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 order 1 tensors, they will simply be referred to as vectors.
- A **matrix** is a two-dimensional array of numbers, such as a collection of vectors This can be vizualized on Figure 1.2b. 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 $\mathbf{X}(i, j)$ woud represent the i^{th} entry of vector j. More generally, entry $(i, j) \in [m] \otimes [n]$ of $\mathbf{X}$ is denoted as $X(i, j)$ or compactly as $\mathbf{x}_{i,j}$. 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

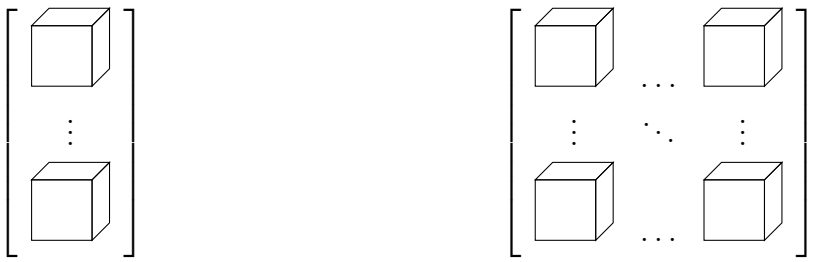
(b) Matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ is a 2-way tensor

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

Figure 1.2: Tensors of orders one, two, and three

If we have a three-dimensional array of numners, then we have a higher-order tensor. Tensors of order 3 or greater are denoted by uppercase mathematical caligraphy letters: $\mathcal{X}$. This can be vizualized on Figure 1.2c. 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 scenatios, the tensor entry $\mathcal{X}(i, j, k)$ would represent 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 $X(i, j, k)$ 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 **uniform**.

As mentioned earlier, any tensor of order greater 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. Figure 1.3 illustrate 4-way and 5-way tensors. There, we can visualize the recursive nature of tensors. A 4-way tensor is really an array of 3-way tensors, in real life that is often used to describe an experiment with three spatial dimensions and one time dimension. Similarly, a 5-way tensor is really a martrix of 3-way tensors, in real life that is often used to describe an experiment with 3 spatial dimensions, one time dimension, and and a certain amount of variables being measured is the fifth fimension. To remember all of this, 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
$\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4 \times n_5}$

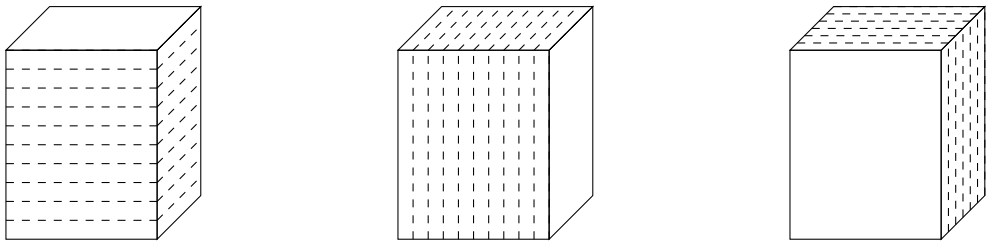
Figure 1.3: Tensors of orders four and five

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 $x_{i_1i_2i_3i_4}$
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 $x_{i_1i_2\cdots i_d}$

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 1.4.



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

Figure 1.4: Two-way slices of a 3-way tensor

The concept of slices is generalizable for higher-order tensors of order greater than or equal to 4, those are called hyperslices. But since that goes beyond the scope of this work, it will not be covered here.

Tensor fibers are the analogs of rows and columns of matrices. The main differene 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:

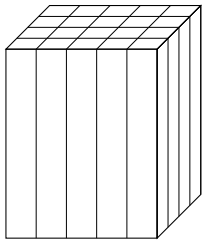
- The **mode-1 fibers** of length m, also known as **column fibers**, range over all indices in the first mode, holding the second and third indices fixed. In other words, there are np column fibers of the form $\mathbf{x}_{:jk} \in \mathbb{R}^m$. This can be vizualized on Figure 1.5a.

- 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 vizualized on Figure 1.5b.

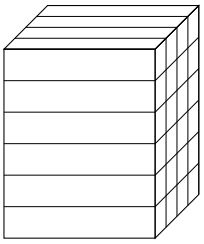
- The **mode-3 fibers** of length m, also known as **tube fibers**, range over all indices in the third mode, holding the first and second indices fixed. In other words, there are mn tube fibers of the form $\mathbf{x}_{ij:} \in \mathbb{R}^p$. This can be vizualized on Figure 1.5c.

1.1.3 Tensor Mode-k Unfoldings

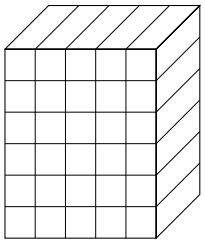
The elements of a tensor can be rearranged to form various matrices in a procedure referred to as **unfolding**, also known as **matricization**. A particular unfolding of



(a) Column Fibers $\mathbf{x}_{:jk}$



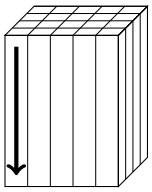
(b) Row Fibers $\mathbf{x}_{i:k}$



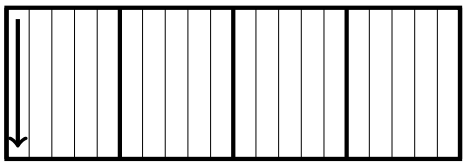
(c) Tube Fibers $\mathbf{x}_{ij:}$

Figure 1.5: Fibers of a 3-way tensor

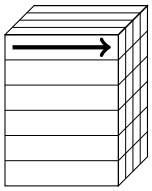
interest is the mode-k unfolding which is defined as a matrix whose columns are the mode-k fibers of that tensors. The notation for a mode-k unfolding of a tensor $\mathcal{X}$ is $\mathbf{X}_{(k)}$. Figure 1.6 illustrates the mode-k unfoldings of a 3-way tensor.



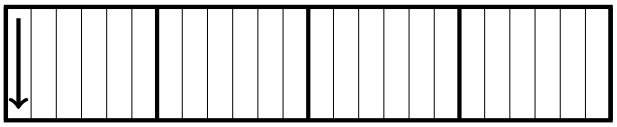
(a) Mode 1 Fibers



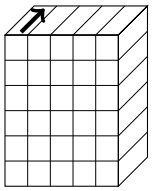
(b) Mode 1 Unfolding $\mathbf{X}_1$



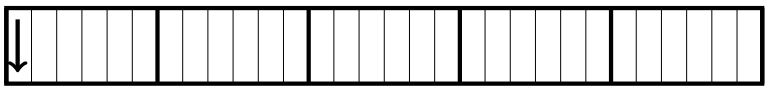
(c) Mode 1 Fibers



(d) Lateral Slices $\mathbf{X}_2$



(e) Mode 3 Fibers



(f) Frontal Slices $\mathbf{X}_3$

Figure 1.6: Unfoldings of a 3-way tensor

1.1.4 Types of Tensor Multiplication

There are several types of Tensor operations, most of which won't be covered here as they go beyond the scope of this work. There are some types of products however that are relevant for us. Since multiplication involves two objects, the purpose of this subsection is journey through the possible types of multiplication increase the number of dimensions of each side of the multiplication one at a time. The first few are trivial and can be explained briefly. Multiplication of two tensors of order zero is trivial, after all you are simply multiplying two scalars. Increasing the dimension of one side of the multiplication while fixing the other gets us a multiplition of an order zero tensor by an order one tensor (a vector). In this product, we are simply *scaling* the entries of the vector by the scalar (hence the name). Moving on, we get to products of two tensors of order 1, i.e. vector-vector products. There are actually two types of products in this scenario the inner product and the outer product which are explained below. Though there are also definition for other types of products for the scenarios that follow, we will refrain from defining those here. As we parse through these types of multiplications, there are two important concepts that arise. The *matching* of the dimensions, and the *contraction* of these matching dimensions, more on that later.

Vector Inner Products

Starting small, the inner product of two same-sized vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ produces a scalar, and is denoted as $\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^\intercal \mathbf{b}$, and is defined as in equation 1.1. A vector norm is defined as the inner product of a vector with itself: $\langle \mathbf{a}, \mathbf{a} \rangle$. The computational complexity of vector inner products is $\mathcal{O}(n)$

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

Vector Outer Products

In contrast to vector inner products are a reductive operation which generates a scalar, vector outer products are an expansive operation which generates a matrix. Vector outer products are defined for multiple vectors, which is why we avoid the common notaion of $\mathbf{a}^\intercal \mathbf{b}$. So we start by showing 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 in equation 1.2. The outer product of two vectors, generates a two-dimensional 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 $\mathcal{O}(mn)$. In general, the computational complexity of d vectors of respective size of $n_1, \cdots, n_d$ is $\mathcal{O}(\prod_{i=1}^{d} n_d)$.

$$\mathbf{C} = \mathbf{a} \circ \mathbf{b} \in \mathbb{R}^{m \times n}, \text{ where } c_{ij} = a_i b_j, \forall (i,j) \in [m] \times [n] \quad (1.2)$$

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

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 in equation 1.3. The computational complexity of the matrix-vector product is $\mathcal{O}(mn)$.

$$\mathbf{y} = \mathbf{A}\mathbf{x} \in \mathbb{R}^m, \text{ where } y_i = \sum_{j=1}^{n} a_{ij} x_j \text{ for all } i \in [m] \quad (1.3)$$

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

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 in equation 1.4. The computational complexity of the matrix-matrix product is $\mathcal{O}(mnp)$. In general, the computational complexity of multiplying to same-sized matrices of size n by n is $\mathcal{O}(n^3)$. A simple and yet noteworthy example can be seen at 1.5, which shows that matrix multiplication is performed by inner product of the rows of $\mathbf{A}$ with

the columns of $\mathbf{B}$ for every row of $\mathbf{A}$, for every column of $\mathbf{B}$.

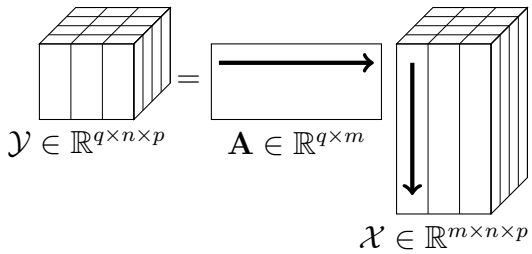
$$\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times n}, \text{ where } c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}, \forall (i,j) \in [m] \otimes [n] \tag{1.4}$$

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

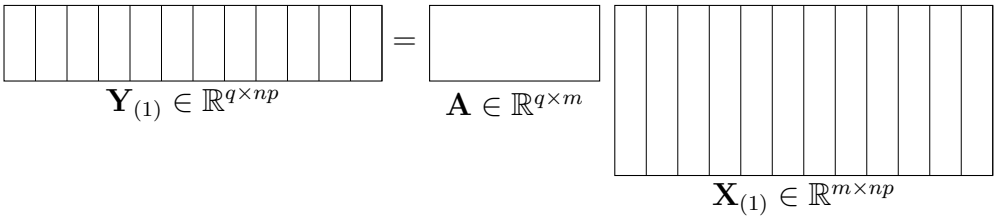
Tensor-Times-Matrix (TTM) Products

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 $\mathbf{Y}_{(k)} = \mathbf{AX}_{(k)}$. 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 fibers of $\mathcal{X}$ by $\mathbf{A}$. Take a look for example at Figure 1.7, in 1.7b we see the notion that a TTM multiply each column fiber of $\mathcal{X}$ by the rows of A, and in 1.7b we see the notion that this is equivalent to unfolding the input and output tensors and performing a regular matrix multiplication.

Theoretically, this is all that TTM is, but in practice things get more difficult. Mode-1 TTMs really are as simple as what the figure conveys, but for other modes the way the tensor is stored in memory has a huge impact on the way we perform



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

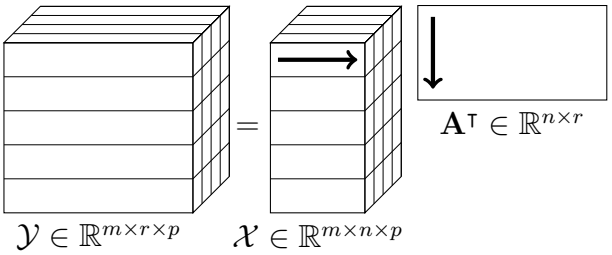


(b) Matrix form.

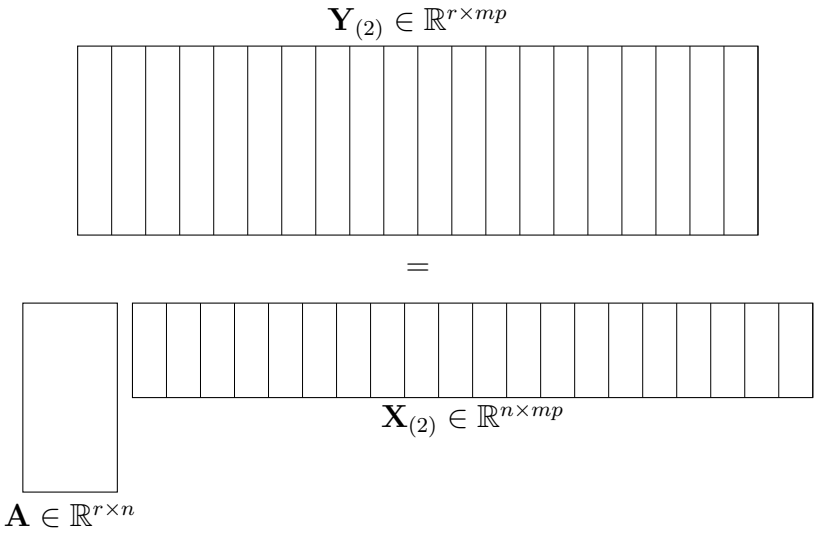
Figure 1.7: Mode-1 TTM (along column fibers)

the TTM. The details of how to do so are omitted in this work, but if you wish to go understand these finer aspects of how TTMs are performed in practice, I recommend you check out [2]. Because of this omission, Figure 1.8 showcases the theoretical way of performing a TTM on mode 2, which isn't how such is performed computationally. Similarly to mode-1 TTM, we can either visualize it in its tensor format in 1.8a as applying the matrix $\mathbf{A}$ to the row fibers of $\mathcal{X}$. Notice how A is transposed here, this is a side effect of TTMs performed in any mode that is not the first and the last of its tensor. If we wish to visualize this TTM as matrix multiplication, the transpose goes away, and Figure 1.8b has the same format as Figure 1.7b.

It is in the matrix-multiplication format that one can understand best the nature of the details of how TTM operations are performed computationally. To provide a



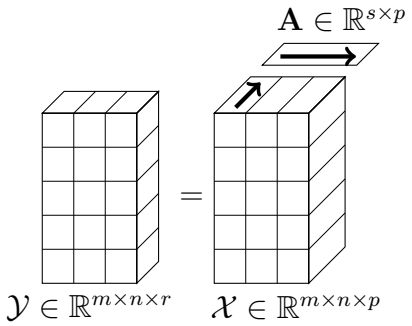
(a) Tensor form: the first row of $\mathbf{A}$ and first mode-2 fiber of $\mathcal{X}$ are emphasized with arrows.



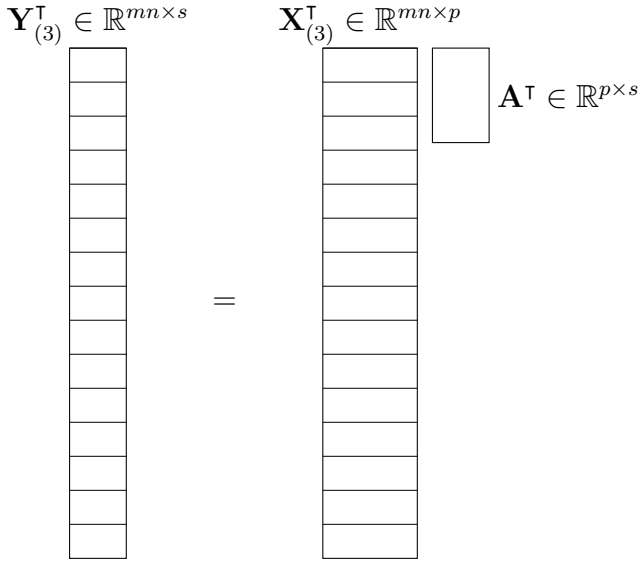
(b) Matrix Form

Figure 1.8: Mode-2 TTM (along row fibers)

small glimpse of it we turn our eyes to the mode 3 TTM in Figure 1.9. Notice that the matrix multiplication format of this TTM in 1.9b have been adapted to have the transpose of all elements involved. This is to avoid any memory movement which is deemed expensive. Again, more details of this can be found on [2]



(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 1.9: Mode-3 TTM (along tube fibers)

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 diving in to this one which is fairly difficult to visualize by the means of drawing as we have been seeing so far, it is worth taking a step back to reflect on the types of multiplication already covered.

Notice how from the very first type of multiplication, we needed the *inner dimensions to match*. For 1D tensors, arrays, 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 $a^\intercal b$ (inner product) or $ab^\intercal$ (outer product). In other words, we could either perform a $1 \times n$ by $n \times 1$ inner product or an $n \times 1$ by $1 \times n$ outer product. The former *contracts* the n vs n dimensions to produce a 1 by 1 scalar, and the latter *contracts* the 1 vs 1 dimensions 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, contract the matching modes of two tensors.

Consider two tensors, $\mathcal{X} \in \mathbb{R}^{m \times n \times p}$ and $\mathcal{X} \in \mathbb{R}^{p \times q \times r}$. 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

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

As mentioned earlier, the nature of the drawings presented so far are not useful for visualizing this form of multiplication, this is where we turn to tensor diagrams. 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.

1.2 Tensor Decompositions

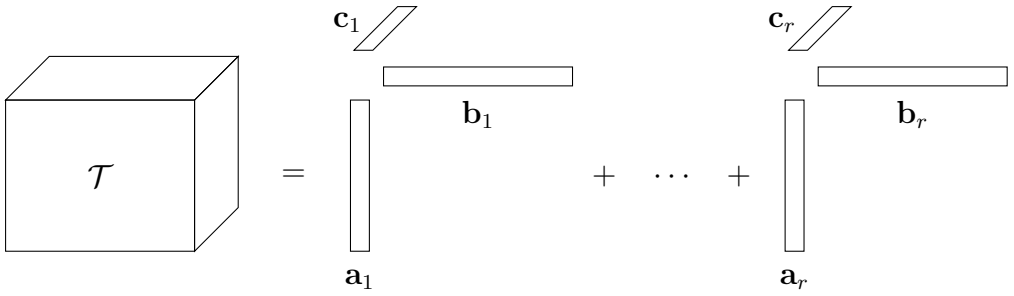
Tensors suffer from the infamous **curse of dimensionality**. This curse exists because as the number of the dimensions of a tensor grows, its storage cost and the cost of operations involving it grows exponentially. This is because the number of entries in a uniform d-way tensor is n^d. Thus, there 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 approximations, though the latter is much more common. 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 two of the most famous ones; The CP Decomposition, and the Tucker Decomposition. As mentioned back in the abstract, this thesis is build upon two projects, each project will focus on one of these decompositions. Furthermore, though it was just said that exact tensor decompositions are rarer, the first project focuses on exact CP decompositions of a special type of tensor. On the other hand, the second project focuses on numeric tensor approximations using the Tucker Decomposition.

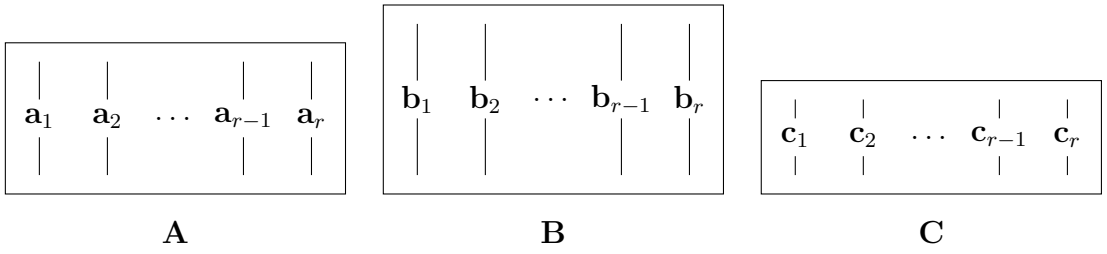
1.2.1 Kruskal Tensors and The CP Decomposition

The CP Decomposition compresses an input tensor into a Kruskal Tensor (KTensor), which is a collection 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. The vectors in each mode come together to form a factor matrix. We can visualize this in the case of a 3-way tensor as shown in Figure 1.10a.

The rank-1 components in their outer-product format are cassually and kindheartdly referred to as chicken feet for their appearance. It is crucial to note that we can move these chicken feet around as long as we keep their corresponding indeces accross all factor matrices together, this fact will be relevant later on.



(a) A 3 way Kruskal Tensor Diagram



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

Figure 1.10: The CP Decomposition

Mathetmatically, given a tensor $\mathcal{T} \in \mathbb{R}^{m \times n \times p}$ and decomposition rank $r \in \mathbb{N}$, The goal of a CP Decomposition is to find factor matrices $\mathbf{A} \in \mathbb{R}^{m \times r}, \mathbf{B} \in \mathbb{R}^{n \times r}, \mathbf{C} \in \mathbb{R}^{p \times r}$ such that

$$t_{ijk} = \sum_{\ell=1}^{r} a_{i\ell} b_{j\ell} c_{k\ell}, \ \forall (i,j,k) \in [m] \times [n] \times [p] \tag{1.6}$$

Or alternatively:

$$\mathcal{T} \approx [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!] = \sum_{\ell=1}^{r} a_\ell \circ b_\ell \circ c_\ell \tag{1.7}$$

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 outer product of $n \times 1$ two vectors. Traditional methods of computing the CP decomposition of a tensor are gradient descent and the newton method. The method used in this work is a variation of the later called damped gauss newton. All these methods often minimize the sum of squares error. The least squares error is shown in 1.8:

$$\|\mathcal{T} - [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!]\|^2 \equiv \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{p} \left(t_{ijk} - \sum_{\ell=1}^{r} a_{i\ell} b_{j\ell} c_{k\ell} \right)^2 \tag{1.8}$$

Thus, the CP optimization problem for a given r is shown in 1.9. This is a non-convex problem.

$$\min_{A} \|\mathcal{T} - [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!]\|^2, \text{ subject to } \mathbf{A} \in \mathbb{R}^{m \times r} \mathbf{B} \in \mathbb{R}^{n \times r} \mathbf{C} \in \mathbb{R}^{p \times r} \tag{1.9}$$

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 1.11.

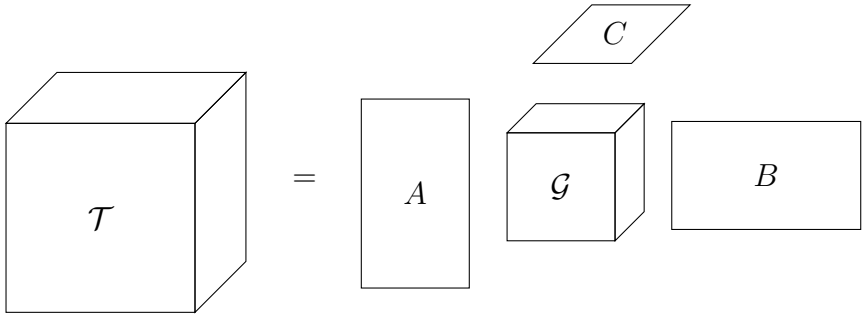


Figure 1.11: A 3-way Tucker Tensor Diagram

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 as seen in 1.10, but if we wish to reconstruct but a single entry then we perform 1.11

$$\mathcal{T} \approx [\![\mathcal{G}; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!] = \mathcal{G} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} \tag{1.10}$$

$$t_{ijk} = \sum_{\alpha=1}^{q} \sum_{\beta=1}^{r} \sum_{\gamma=1}^{s} g_{\alpha\beta\gamma} \cdot a_{i\alpha} b_{j\beta} c_{i\gamma}, \forall (i,j,k) \in [m] \otimes [n] \otimes [p] \tag{1.11}$$

The traditional methods are HOSVD (Higher Order Singular Value Decomposition) and STHOSVD (Sequentially Truncated Singular Value Decomposition). Though

HOOI (Higher Order Orthogonal Iteration) is not as traditional, it will be the protagonist for this work. We describe these algorithms in Section 3.1

The CP Decomposition

Despite the tile, the focus of this project are not CP Decompositions. Rather, their active use in searching for fast matrix multiplication algorithms. There was a glimpse of these types of algorithms in Section 1.1.4, but the rabbit hole goes deeper than that. Therefore, the use of CP Decompositons of a special tensor to discover new fast matrix multiplication algorithms is left for Section 2.1.2, and Section 2.1.1 starts this chapter with some much-needed background about Matrix Multiplication Algorithms.

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 fact, Equation (1.5) demonstrates how to multiply two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{2\times 2}$. Matrices $\mathbf{A}$ and $\mathbf{B}$ don't necessarily need to be of size 2 by 2, as long as they are of a size that is a multiple of 2, we can perform such matrix multiplication using a recursive algorithm

like the one on Algorithm 1. This is the only time such algorithm is presented using the pseudocode format, for the sake of simplicity algorithms will be showcased in the format displayed in 2.1.

Algorithm 1 MatMul

function $\mathbf{C} = \text{MatMul}(\mathbf{A}, \mathbf{B})$
  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

Classic 2 by 2 Matrix Multiplication Algorithm

$$\begin{aligned}
\mathbf{M}_1 &= \mathbf{A}_{11} \cdot \mathbf{B}_{11} \\
\mathbf{M}_2 &= \mathbf{A}_{12} \cdot \mathbf{B}_{21} \\
\mathbf{M}_3 &= \mathbf{A}_{11} \cdot \mathbf{B}_{12} \\
\mathbf{M}_4 &= \mathbf{A}_{12} \cdot \mathbf{B}_{22} \\
\mathbf{M}_5 &= \mathbf{A}_{21} \cdot \mathbf{B}_{11} \\
\mathbf{M}_6 &= \mathbf{A}_{22} \cdot \mathbf{B}_{21} \\
\mathbf{M}_7 &= \mathbf{A}_{21} \cdot \mathbf{B}_{12} \\
\mathbf{M}_8 &= \mathbf{A}_{22} \cdot \mathbf{B}_{22} \\
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_2 \\
\mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_4 \\
\mathbf{C}_{21} &= \mathbf{M}_5 + \mathbf{M}_6 \\
\mathbf{C}_{22} &= \mathbf{M}_7 + \mathbf{M}_8
\end{aligned} \tag{2.1}$$

This algorithm involves 8 multiplications and 4 additions of matrices of size $n/2$. Therefore, our computational complexity becomes $T(n) = 8T(n/2) + O(n^2) = O(n^{\log_2 8}) = O(n^3)$. This computational cost can be decresed by carefully manipulating our 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 is showcased in 2.2. His algorithm involves 7 multiplications and 18 additions, but recall that additions are computationally less expensive than multiplications. The computational complexity of Strassen's algorithm is $T(n) = 7T(n/2) + O(n^2) = O(n^{\log_2 7}) \approx O(n^{2.81})$.

Classic Strassen's Algorithm

$$
\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22}) \cdot (\mathbf{B}_{11} + \mathbf{B}_{22}) \\
\mathbf{M}_2 &= (\mathbf{A}_{12} + \mathbf{A}_{22}) \cdot \mathbf{B}_{11} \\
\mathbf{M}_3 &= \mathbf{A}_{11} \cdot (\mathbf{B}_{21} - \mathbf{B}_{22}) \\
\mathbf{M}_4 &= \mathbf{A}_{22} \cdot (\mathbf{B}_{12} - \mathbf{B}_{11}) \\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{21}) \cdot \mathbf{B}_{22} \\
\mathbf{M}_6 &= (\mathbf{A}_{12} - \mathbf{A}_{11}) \cdot (\mathbf{B}_{11} + \mathbf{B}_{21}) \\
\mathbf{M}_7 &= (\mathbf{A}_{21} - \mathbf{A}_{22}) \cdot (\mathbf{B}_{12} + \mathbf{B}_{22}) \\
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{21} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}
\tag{2.2}
$$

Strassen's algorithm can be rearranged; we can modify the additions and multiplies to get another 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. This permutted version will be relevant for Section 2.2 as it contains a special structure that is hard to visualize in the Equation (2.2). Notice how the two algorithms are the same, except that $\mathbf{M}_3$ became $\mathbf{M}_6$, $\mathbf{M}_4$ became $\mathbf{M}_3$, and $\mathbf{M}_6$ became $\mathbf{M}_4$, and the additions in $\mathbf{C}$ changed respectively. It is explained in Section 2.2.1 that this is due to a change in the columns of a KTensor decomposition.

Permuted Strassen's Algorithm

$$\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22}) \cdot (\mathbf{B}_{11} + \mathbf{B}_{22}) \\
\mathbf{M}_2 &= (\mathbf{A}_{12} + \mathbf{A}_{22}) \cdot \mathbf{B}_{11} \\
\mathbf{M}_3 &= \mathbf{A}_{22} \cdot (\mathbf{B}_{12} - \mathbf{B}_{11}) \\
\mathbf{M}_4 &= (\mathbf{A}_{21} - \mathbf{A}_{22}) \cdot (\mathbf{B}_{12} + \mathbf{B}_{22}) \\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{21}) \cdot \mathbf{B}_{22} \\
\mathbf{M}_6 &= \mathbf{A}_{11} \cdot (\mathbf{B}_{21} - \mathbf{B}_{22}) \\
\mathbf{M}_7 &= (\mathbf{A}_{12} - \mathbf{A}_{11}) \cdot (\mathbf{B}_{11} + \mathbf{B}_{21}) \\
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_3 - \mathbf{M}_4 - \mathbf{M}_6 \\
\mathbf{C}_{12} &= \mathbf{M}_2 + \mathbf{M}_3 \\
\mathbf{C}_{21} &= \mathbf{M}_5 + \mathbf{M}_6 \\
\mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_5 + \mathbf{M}_7
\end{aligned} \tag{2.3}$$

It is at this point that an important distinction must be made; what is considered a new algorithm? Both Strassen's algorithm and its permutation are considered Strassen's algorithm because they both involve 7 multiplications. This is because the term *Strassen's algorithm* has become an all-encapsulating term for any 2 by 2 matrix multiplication algorithm that uses 7 multiplications instead of the classic 8. For example, take a look at Equation (2.4). It is no longer just a permutted version of Equation (2.2) as this onw has 24 matrix additions as opposed to the original's 18. Again, since matrix addition is not the bottleneck, both of these algorithms have

the same computational complexity. The algorithm below is still called Strassen's algorithm for the reasons mentioned previously. Furthermore, this variant version will also be relevant in Section 2.2 as it contains the same srtructure we will explore later on, but slighly different. It has been proven that there are no possible fast matrix multiplication algorithms with 6 multiplications.

Variant Strassen's Algorithm

$$
\begin{aligned}
\mathbf{M}_1 &= \mathbf{A}_{11} \cdot \mathbf{B}_{11} \\
\mathbf{M}_2 &= (\mathbf{A}_{12} + \mathbf{A}_{22}) \cdot (\mathbf{B}_{12} + \mathbf{B}_{22}) \\
\mathbf{M}_3 &= (\mathbf{A}_{22} - \mathbf{A}_{21}) \cdot (\mathbf{B}_{22} - \mathbf{B}_{21}) \\
\mathbf{M}_4 &= (\mathbf{A}_{21} - \mathbf{A}_{12} - \mathbf{A}_{22}) \cdot (\mathbf{B}_{21} - \mathbf{B}_{12} - \mathbf{B}_{22}) \\
\mathbf{M}_5 &= (-\mathbf{A}_{12}) \cdot (-\mathbf{B}_{21}) \\
\mathbf{M}_6 &= (\mathbf{A}_{11} - \mathbf{A}_{12} + \mathbf{A}_{21} - \mathbf{A}_{22}) \cdot (-\mathbf{B}_{12}) \\
\mathbf{M}_7 &= (-\mathbf{A}_{21}) \cdot (\mathbf{B}_{11} - \mathbf{B}_{12} + \mathbf{B}_{21} - \mathbf{B}_{22}) \\
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_5 \\
\mathbf{C}_{22} &= \mathbf{M}_4 - \mathbf{M}_3 + \mathbf{M}_5 + \mathbf{M}_6 \\
\mathbf{C}_{22} &= \mathbf{M}_2 - \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{22} &= \mathbf{M}_2 + \mathbf{M}_3 - \mathbf{M}_4 + \mathbf{M}_5
\end{aligned}
\tag{2.4}
$$

Thus, a question arises, how many possibilities are there? We could search for different solutions that all have the same number of multiplications by performing either an eshaustive search or an optimization search on each parameter involved in matrix multiplication as seen in 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 we can move on to how to search for these algorithms non-exhaustively, we turn to the protagonist of this project.

Exhaustive Search of Fast MatMul Algorithms

$$
\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}) \\
\\
\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}
\tag{2.5}
$$

2.1.2 The Matrix Multiplication Tensor

There is a specific type of tensor that is crucial to this project: the Matrix Multiplication Tensor. Matrix multiplication can be performed in tensor format using this tensor as visualized in Figure 2.1. If we wish to multiply two 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. In order to do so, we vectorize $\mathbf{A}$ and $\mathbf{B}$ and multiply them through 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. $\mathbf{C}^\intercal$). 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

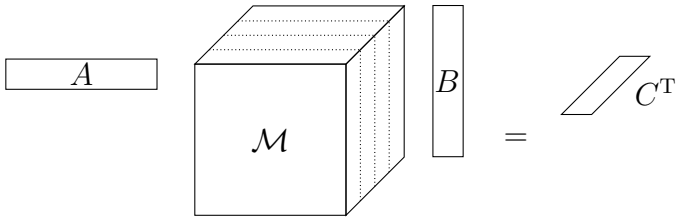


Figure 2.1: Matrix Multiplication in Tensor Format

$\mathcal{M} \in \mathbb{R}^{n^2 \times n^2 \times n^2}$.

Algorithm 2 Forming the Matrix Multiplication Tensor

```
function T = MatMul-Tensor(m, n, p)
    T = zeros(mn, np, mp)                          ▷ Initialize Tensor
    for i = 1 : m do
        for i = 1 : m do
            for i = 1 : m do
                T(mj + i, nk + j, pi + k) = 1
            end for
        end for
    end for
end function
```

It is at this point we return to 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. It turns out, that if we decompose the matrix multiplication tensor using the CP Decomposition, there is a hidden fast matrix multiplication algorithm embedded in the components of the CP decomposition. In fact, the number of components r corresponds to the number of multiplications in the algorithm. This in return results in two different implications (could argue that they mean the same thing in two different directions but we seperate these implications for the sake of understanding). The first implication is that given an algorithm, say

one of the $3^8 4$ 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 matmul tensor. If $\|\mathcal{M} - \hat{\mathcal{M}}\| = 0$, then we have a valid algorithm, there are easier ways to check for the validity of an algorithm, but this helps with forming the search equation. Before continuing with the second implication, we must understand how to visualize the hidden algorithm in the factor matrices of a KTensor.

Below is the original Strassen's algorithm with two representations, the one we have seen before and the KTensor representation. The way to interpret the algorithm on the right, is that each horizontal lines separate the factor matrices, therefore just like in Figure 1.10a the factor matrices $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ are separated by the horizontal lines, the columns of the factor matrices represent the chicken feet components. If you pay close attention you can see 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.

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

	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$	$\mathbf{M}_4$	$\mathbf{M}_5$	$\mathbf{M}_6$	$\mathbf{M}_7$
$\mathbf{A}_{11}$	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

The other implication, is that we can decompose the matmul tensor with specified CP rank (as in Equation (1.9)) through an optimization algorithm to search for fast matrix multiplication algorithms.

2.1.3 Damped Gauss Newton Optimization for CP Decompositions

Recall from Equation (1.8) that we perform a CP decomposition with the least squares error of our approximation. We will do so using optimization methods for reasons that will become clear soon. Optimization methods work with vector input, so for our purposes let

$$\mathbf{v} = \text{vec}\left(\begin{bmatrix} \mathbf{A} \\ \mathbf{B} \\ \mathbf{C} \end{bmatrix}\right) = \begin{bmatrix} \text{vec}(\mathbf{A}) \\ \text{vec}(\mathbf{B}) \\ \text{vec}(\mathbf{C}) \end{bmatrix} \in \mathbb{R}^{3nr}$$

Furthermore, let $\phi : \mathbb{R}^n \to \mathbb{R}^m$ be the nonlinear function.

$$\phi(\mathbf{v}) = \text{vec}(\mathcal{M} - [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!]) \tag{2.6}$$

Thus, Equation (1.8) becomes

$$\min_{A,B,C} f(\mathbf{v}) = \frac{1}{2}\|\phi(\mathbf{v})\|^2 \tag{2.7}$$

With that out of the way we must choose an optimization method. Gradient is too simple, taking one step further we reach Newton's method. Newton's method

gets the search direction by using Newton's equation

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

where $\mathbf{d}_k$ is our approximation. We dislike the Hessian here because it is computationally expensive and difficult to derive mathematicall (perhaps not now but for the cylic part). Thus, we prefer the Gauss-Newton method which approximates the Hessian with $\mathbf{J}^\intercal\mathbf{J}$ where $\mathbf{J} : \mathbb{R}^n \to \mathbb{R}^{n\times m}$ is the jacobian of ϕ:

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

But even then, the Gauss-Newton matrix $\mathbf{J}^\intercal\mathbf{J}$ can be singular, we add a damping parameter, $\lambda\mathbf{I}$, to enforce positive definiteness. Thus, we reach the damped Gauss-Newton Method:

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

. At this point, we have two tasks at hand. Given our $\mathbf{v}, f(\mathbf{v})$ and $\phi(\mathbf{v})$, we must derive the gradient of f for the right-hand side and the jacobian of ϕ for the left-hand side (when really we just want to know how to apply it to a vector). The gradient is easy.

$$\nabla f = \operatorname{vec}\left(\begin{bmatrix} \frac{\partial f}{\partial \mathbf{A}} \\ \frac{\partial f}{\partial \mathbf{B}} \\ \frac{\partial f}{\partial \mathbf{C}} \end{bmatrix}\right) = \begin{bmatrix} \frac{\partial f}{\partial \operatorname{vec}(\mathbf{A})} \\ \frac{\partial f}{\partial \operatorname{vec}(\mathbf{B})} \\ \frac{\partial f}{\partial \operatorname{vec}(\mathbf{C})} \end{bmatrix} \in \mathbb{R}^{3nr} \tag{2.8}$$

Where each partial derivative is defined as:

$$\begin{array}{rcl} \frac{\partial f}{\partial \mathbf{A}} & = & -\mathbf{M}_{(1)}(\mathbf{C} \odot \mathbf{B}) + \mathbf{A}(\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B}) \\ \frac{\partial f}{\partial \mathbf{B}} & = & -\mathbf{M}_{(1)}(\mathbf{C} \odot \mathbf{A}) + \mathbf{B}(\mathbf{C}^\intercal\mathbf{C} * \mathbf{A}^\intercal\mathbf{A}) \\ \frac{\partial f}{\partial \mathbf{C}} & = & -\mathbf{M}_{(1)}(\mathbf{B} \odot \mathbf{A}) + \mathbf{C}(\mathbf{B}^\intercal\mathbf{B} * \mathbf{A}^\intercal\mathbf{A}) \end{array} \tag{2.9}$$

The Jacobian is not so easy.

$$\mathbf{J} = [\mathbf{J}_\mathbf{A} + \mathbf{J}_\mathbf{B} + \mathbf{J}_\mathbf{C}] \in \mathbb{R}^{n^3 \times 3nr} \tag{2.10}$$

Where

$$\begin{array}{r} \mathbf{J}_\mathbf{A} \equiv \frac{\partial \phi}{\partial \mathrm{vec}(\mathbf{A})} = (\mathbf{C} \odot \mathbf{B}) \otimes \mathbf{I} \in \mathbb{R}^{n^3 \times nr} \\ \mathbf{J}_\mathbf{B} \equiv \frac{\partial \phi}{\partial \mathrm{vec}(\mathbf{B})} = \mathbf{\Pi}_2^\intercal(\mathbf{C} \odot \mathbf{A}) \otimes \mathbf{I} \in \mathbb{R}^{n^3 \times nr} \\ \mathbf{J}_\mathbf{C} \equiv \frac{\partial \phi}{\partial \mathrm{vec}(\mathbf{C})} = \mathbf{\Pi}_3^\intercal(\mathbf{B} \odot \mathbf{A}) \otimes \mathbf{I} \in \mathbb{R}^{n^3 \times nr} \end{array} \tag{2.11}$$

such that that Π_k is the tensor perfect shuffle matrix, such that $\mathrm{vec}(\mathcal{X}) = \Pi_k$ $\mathrm{vec}(\mathbf{X}_{(k)})$, and Π_1 is not written explicitly because it is the identity matrix. We consider fast computation 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 without computing any explicit Kronecker or Khatri-Rao products. From Equation (2.10) we have that the block structure of $\mathbf{J}^\intercal\mathbf{J}$ is

$$\mathbf{J}^\intercal\mathbf{J} = \begin{bmatrix} \mathbf{J}_\mathbf{A}^\intercal\mathbf{J}_\mathbf{A} & \mathbf{J}_\mathbf{A}^\intercal\mathbf{J}_\mathbf{B} & \mathbf{J}_\mathbf{A}^\intercal\mathbf{J}_\mathbf{C} \\ \mathbf{J}_\mathbf{B}^\intercal\mathbf{J}_\mathbf{A} & \mathbf{J}_\mathbf{B}^\intercal\mathbf{J}_\mathbf{B} & \mathbf{J}_\mathbf{B}^\intercal\mathbf{J}_\mathbf{C} \\ \mathbf{J}_\mathbf{C}^\intercal\mathbf{J}_\mathbf{A} & \mathbf{J}_\mathbf{C}^\intercal\mathbf{J}_\mathbf{B} & \mathbf{J}_\mathbf{C}^\intercal\mathbf{J}_\mathbf{C} \end{bmatrix}$$

then

$$\mathbf{J}^\intercal\mathbf{J}\mathbf{v} = \begin{bmatrix} \mathbf{J}_\mathbf{A}^\intercal\mathbf{J}_\mathbf{A}\text{vec}(\mathbf{A}) + \mathbf{J}_\mathbf{A}^\intercal\mathbf{J}_\mathbf{B}\text{vec}(\mathbf{B}) + \mathbf{J}_\mathbf{A}^\intercal\mathbf{J}_\mathbf{C}\text{vec}(\mathbf{C}) \\ \mathbf{J}_\mathbf{B}^\intercal\mathbf{J}_\mathbf{A}\text{vec}(\mathbf{A}) + \mathbf{J}_\mathbf{B}^\intercal\mathbf{J}_\mathbf{B}\text{vec}(\mathbf{B}) + \mathbf{J}_\mathbf{B}^\intercal\mathbf{J}_\mathbf{C}\text{vec}(\mathbf{C}) \\ \mathbf{J}_\mathbf{C}^\intercal\mathbf{J}_\mathbf{A}\text{vec}(\mathbf{A}) + \mathbf{J}_\mathbf{C}^\intercal\mathbf{J}_\mathbf{B}\text{vec}(\mathbf{B}) + \mathbf{J}_\mathbf{C}^\intercal\mathbf{J}_\mathbf{C}\text{vec}(\mathbf{C}) \end{bmatrix}$$

Which actually becomes

$$\mathbf{J}^\intercal\mathbf{J}\mathbf{v} = \begin{bmatrix} \text{vec}(\bar{\mathbf{A}}(\mathbf{B}^\intercal\mathbf{B} * \mathbf{C}^\intercal\mathbf{C}) + \mathbf{A}(\bar{\mathbf{B}}^\intercal\mathbf{B} * \mathbf{C}^\intercal\mathbf{C}) + \mathbf{A}(\mathbf{B}^\intercal\mathbf{B} * \bar{\mathbf{C}}^\intercal\mathbf{C})) \\ \text{vec}(\mathbf{B}(\bar{\mathbf{A}}^\intercal\mathbf{A} * \mathbf{C}^\intercal\mathbf{C}) + \bar{\mathbf{B}}(\mathbf{A}^\intercal\mathbf{A} * \mathbf{C}^\intercal\mathbf{C}) + \mathbf{B}(\mathbf{A}^\intercal\mathbf{A} * \bar{\mathbf{C}}^\intercal\mathbf{C})) \\ \text{vec}(\mathbf{C}(\bar{\mathbf{A}}^\intercal\mathbf{A} * \mathbf{B}^\intercal\mathbf{B}) + \mathbf{C}(\mathbf{A}^\intercal\mathbf{A} * \bar{\mathbf{B}}^\intercal\mathbf{B}) + \bar{\mathbf{C}}(\mathbf{A}^\intercal\mathbf{A} * \mathbf{B}^\intercal\mathbf{B})) \end{bmatrix}$$

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 K_prev to be a cell of length 3 of $n^2 \times r$ matrices
for $i = 1 : MaxIters$ **do**
f $\longleftarrow \frac{1}{2}\|\mathcal{M} - \mathcal{K}\|$ ▷ Compute Function Value
$\nabla\mathbf{f} \longleftarrow [\text{vec}\left(\frac{\partial f}{\partial \mathbf{A}}\right) \text{vec}\left(\frac{\partial f}{\partial \mathbf{B}}\right) \text{vec}\left(\frac{\partial f}{\partial \mathbf{C}}\right)]^\intercal$ ▷ Compute Gradient
$S \longleftarrow$ Solution to$(\mathbf{J}^T\mathbf{J} + \lambda I)K = -\nabla\mathbf{f}$
while *Goldstein Conditions Are Satisfied* **do**
K $\longleftarrow$ K_prev + αS
$\text{f}_{new} \longleftarrow \frac{1}{2}\|\mathcal{M} - \mathcal{K}\|$
$\alpha \longleftarrow \alpha/2$
end while
if f - $\text{f}_{new} < \epsilon$ **then**
break
end if
end for
end function

2.2 Cyclic Invariance

We can reduce our search space in our search for fast matrix multiplication algorithms by levevering **cyclic invariance**. Cyclic invariance is an added structure in matrix multiplication algorithms that reduce the number of variables of the CP Decomposition optimization problem for the matrix multiplication tensor.

2.2.1 Cyclic Invariant Matrix Multiplication Algorithms

Recall that we can permute and rearrange the multiplications and additions of Strassen's agorithms to obtain variantions. Some of these variantions can be cyclic invariant. Below is one of these Strassen's variant algorithm next to the original. Both of these algorithms have already been introduced back in Section 2.1.1. The algorithm on the left is Equation (2.2) and the one on the right is Equation (2.3). 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 $\mathbf{M}_6$, $\mathbf{M}_4$ became $\mathbf{M}_3$, and $\mathbf{M}_6$ became $\mathbf{M}_4$, which force the additions in $\mathbf{C}$ changed accordingly and (2) the permutted 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 it always appears 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

	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$	$\mathbf{M}_4$	$\mathbf{M}_5$	$\mathbf{M}_6$	$\mathbf{M}_7$
$\mathbf{A}_{11}$	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

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 $r_s = r - 3r_c$. 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 both of these algorithms before in Section 2.1.1. The one on the left is Equation (2.3) and the one on the right is Equation (2.4). Recall that the one on the left has 18 matrix additions and the one

on the right has 24. We can easily count such by counting the number of non zero entries in our factor matrices.

	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$	$\mathbf{M}_4$	$\mathbf{M}_5$	$\mathbf{M}_6$	$\mathbf{M}_7$
$\mathbf{A}_{11}$	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

	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$	$\mathbf{M}_4$	$\mathbf{M}_5$	$\mathbf{M}_6$	$\mathbf{M}_7$
$\mathbf{A}_{11}$	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

Recall that our factor matrices correspond to the components of the KTensor as seen in Figure 1.10b. The same factor matrices with ths cyclic invariant structure imposed on them now look like Figure 2.2. That means we can now adjust Figure 1.10a to look like Figure 2.3.

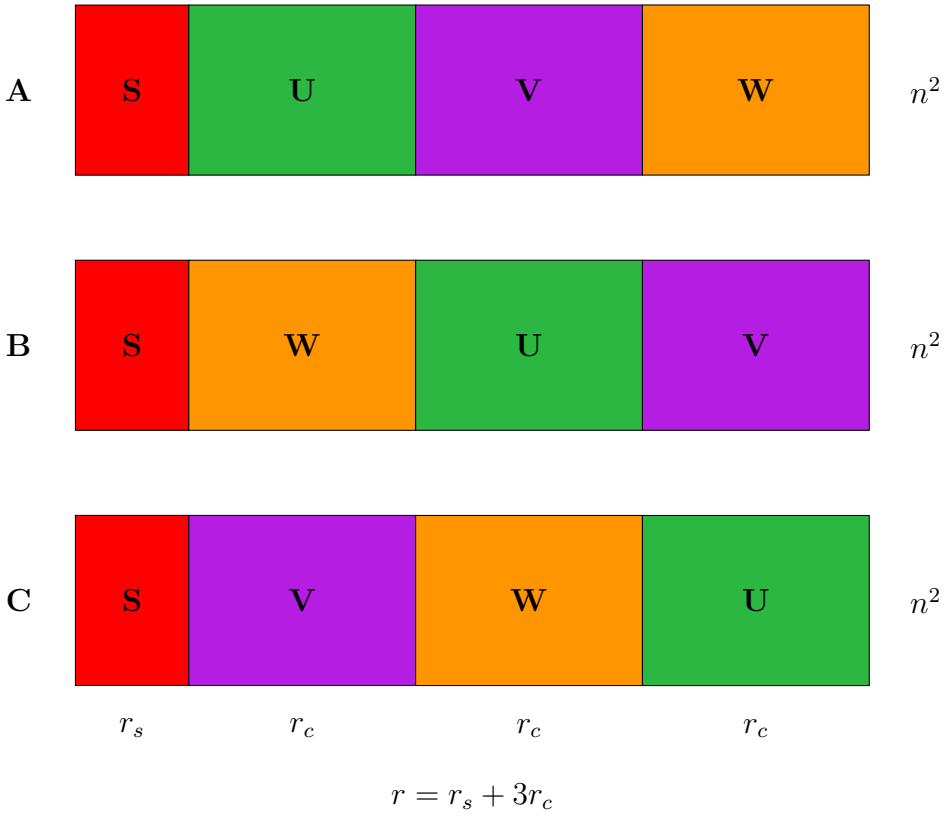


Figure 2.2: Cyclic Invariance in a KTensor

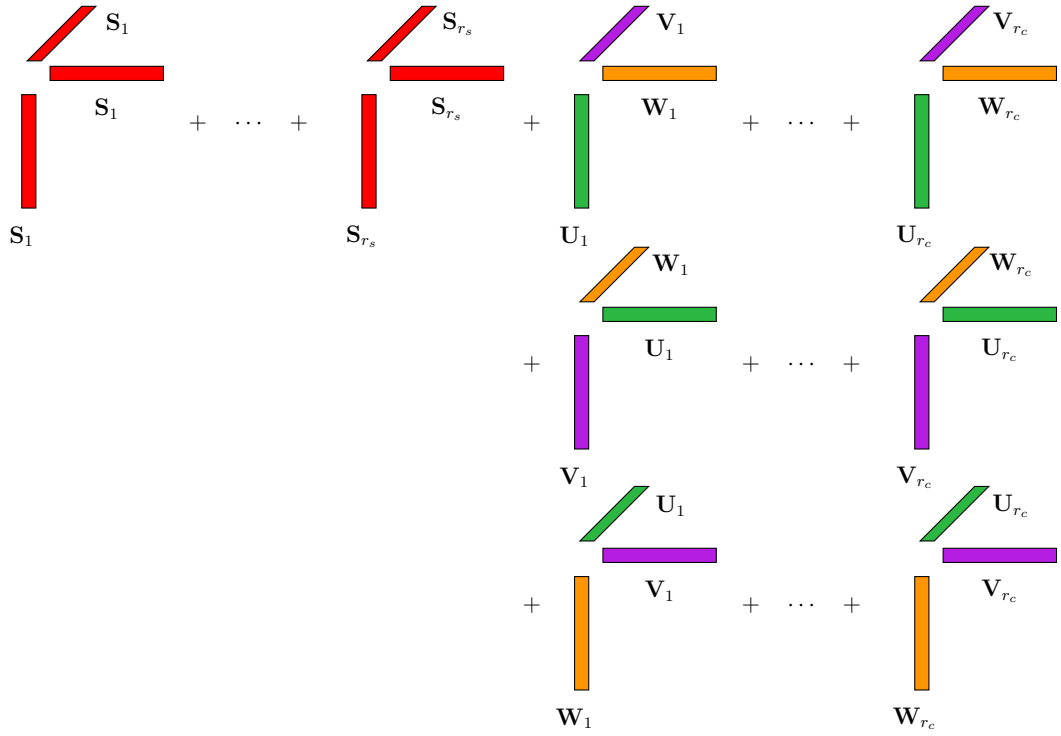


Figure 2.3: CP Decomposition Diagram with Cyclic Invariant Structure

2.2.2 Adapting CP_DGN to Cyclic Invariance

We must now adapt Section 2.1.2 to our cyclic invariance structure. Now we have the imposed structure on any CP Decomposition of the matrix multiplication tensor.

$$\begin{aligned}\mathbf{A} &= [\ \mathbf{S}\quad \mathbf{U}\quad \mathbf{V}\quad \mathbf{W}\ ]\\ \mathbf{B} &= [\ \mathbf{S}\quad \mathbf{W}\quad \mathbf{U}\quad \mathbf{V}\ ]\\ \mathbf{C} &= [\ \mathbf{S}\quad \mathbf{V}\quad \mathbf{W}\quad \mathbf{U}\ ]\end{aligned}$$

That means Equation (1.8) becomes Equation (2.12) in accordance with Figure 2.3.

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

In order for that to be the case, we must change our $\mathbf{v}$ such that

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

The Gradient becomes

$$\nabla f = \begin{bmatrix}\mathsf{vec}(\frac{\partial f}{\partial \mathbf{S}})\\ \mathsf{vec}(\frac{\partial f}{\partial \mathbf{U}})\\ \mathsf{vec}(\frac{\partial f}{\partial \mathbf{V}})\\ \mathsf{vec}(\frac{\partial f}{\partial \mathbf{W}})\end{bmatrix} \in \mathbb{R}^{(m+n+p)r} \tag{2.13}$$

Where each partial derivative is defined as:

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

Similarly, our jacobian becomes

$$\mathbf{J} = [\mathbf{J_S} + \mathbf{J_U} + \mathbf{J_V} + \mathbf{J_W}] \in \mathbb{R}^{n^3 \times nr} \tag{2.15}$$

Where

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

However, just like in Section 2.1.3, we don't care as much about the explicit jacobian as we do about how to apply $\mathbf{J}^\intercal\mathbf{J}$ to a vector.

$$\mathbf{J}^\intercal\mathbf{J} \cdot \text{vec}(\mathbf{K}) = \begin{bmatrix} \mathbf{J_S}^\intercal\mathbf{J_S} & \mathbf{J_S}^\intercal\mathbf{J_U} & \mathbf{J_S}^\intercal\mathbf{J_V} & \mathbf{J_S}^\intercal\mathbf{J_W} \\ \mathbf{J_U}^\intercal\mathbf{J_S} & \mathbf{J_U}^\intercal\mathbf{J_U} & \mathbf{J_U}^\intercal\mathbf{J_V} & \mathbf{J_U}^\intercal\mathbf{J_W} \\ \mathbf{J_V}^\intercal\mathbf{J_S} & \mathbf{J_V}^\intercal\mathbf{J_U} & \mathbf{J_V}^\intercal\mathbf{J_V} & \mathbf{J_V}^\intercal\mathbf{J_W} \\ \mathbf{J_W}^\intercal\mathbf{J_S} & \mathbf{J_W}^\intercal\mathbf{J_U} & \mathbf{J_W}^\intercal\mathbf{J_V} & \mathbf{J_W}^\intercal\mathbf{J_W} \end{bmatrix} \begin{bmatrix} \text{vec}(\mathbf{K_S}) \\ \text{vec}(\mathbf{K_U}) \\ \text{vec}(\mathbf{K_V}) \\ \text{vec}(\mathbf{K_W}) \end{bmatrix} \tag{2.17}$$

The equations for each entry of the above matrix-vector product can be found below:

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

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

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

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

Then, we reach our algorithm

Algorithm 4 Cyclic Invariant CP Damped Gauss-Newton

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 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
for $i = 1 : MaxIters$ **do**
f $\longleftarrow \frac{1}{2}\|\mathcal{M} - [\![\mathbf{S}, \mathbf{S}, \mathbf{S}]\!] - [\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!] - [\![\mathbf{W}, \mathbf{U}, \mathbf{V}]\!] - [\![\mathbf{V}, \mathbf{W}, \mathbf{U}]\!]\|$
$\nabla \mathbf{f} \longleftarrow [\text{vec}\left(\frac{\partial f}{\partial \mathbf{S}}\right) \text{vec}\left(\frac{\partial f}{\partial \mathbf{U}}\right) \text{vec}\left(\frac{\partial f}{\partial \mathbf{V}}\right) \text{vec}\left(\frac{\partial f}{\partial \mathbf{W}}\right)]^T$
$S \longleftarrow$ Solution to $(\mathbf{J}^T\mathbf{J} + \lambda I)\mathbf{K} = -\nabla \mathbf{f}$
while *Goldstein Conditions Are Satisfied* **do**
K $\longleftarrow$ K_prev + αS
f $_{new} \longleftarrow$ Compute Function Value
$\alpha \longleftarrow \alpha/2$
end while
if f - f $_{new} < \epsilon$ **then**
break
end if
end for
end function

2.3 Further Structure in Matrix Multiplication Algorithms

JP: This is ongoing work. Essentially, we are searching for additional structure in the solutions we have found. I will try to summarize the main ideas in the talk as well as in this section before I graduate, but realistic the precise writing of this section won't be in its full version before I graduate

The Tucker Decomposition

The rank of a Tucker decomposition, given by the size of $\mathcal{G}$, directly impacts the amount of compression, so the choice of rank is consequential. If the decomposition's ranks are large, then great accuracy is achieved at the cost of poor compression. If the decomposition's ranks are small, then great compression is achieved at the cost of poor accuracy. This is known as the tensor decomposition trade-off as seen in Figure 3.1. It is applied to all forms of decompositions, but it is specially relevant here.

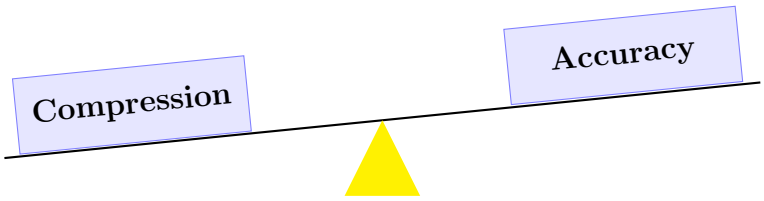


Figure 3.1: The Tensor Decomposition Trade-Off

To know compression beforehand, we specify 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 know the ranks beforehand, this is called the **rank-**

specified formulation, and with it, we also know the compression ratio before hand as seen in Equation (3.1) for the 3-way case. In this formulation, we cannot say in advance what the accuracy will be.

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

To know accuracy beforehand, we specify the maximum relative error threshold ϵ of the tucker approximation. The 3-way case of the relative error can be seen in Equation (3.2). This is called the **error-specified** formulation, where we cannot say in advance what the compression will be.

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

We now go on a journey to understand two of the main Tucker Decomposition algorithms; STHOSVD and HOOI. STHOSVD is the more popular of the two for many reasons, but one of them is that the algorithm works in both rank- and error-specified formulation whereas HOOI, on the other hand, can only work in the rank-specified formulation.

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 STHOSVD

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 STHOSVD algorithm. This method approximately solves either Eq. (3.3) or Eq. (3.4) by unfolding the k^{th} mode of the input tensor, computing its leaft 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 **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 [3] 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 $\mathbf{G}_{(k)}\mathbf{G}_{(k)}$.

Algorithm 5 STHOSVD

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

3.1.2 Classic HOOI

Our protagonist is given in Algorithm 6 and is an alternative method for solving the rank-specified formulation of the Tucker approximation problem [4, 5, 6]. HOOI is a block coordinate descent method and so it requires initial factor matrices. Historically, the output factor matrices of STHOSVD have been used as input factor matrices

for the HOOI algorithm, as the latter has often been as an aditional algorithm to just cheaply clean up the error. However, random factor matrices can be used and generally no more than two iterations are requireed to get a good approximation, often only one iteration is enough to get a descent one.

HOOI iteratively updates each factor matrix by performing a TTM with all but the k^{th} factor matrix to obtain an intermediate tensor $\mathcal{Y}$ and computing the LLSV of $\mathbf{Y}_{(k)}$. The core tensor $\mathcal{G}$ can be computed oce, at the end, or at the end of every iteration in order to computate a per-iteration approximation error.

Algorithm 6 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 $U_{1:d}$ randomly
  $\mathcal{G} \leftarrow \mathcal{X}$
  for Maximum Number of Iterations **do**
    for $k = 1, \ldots, d$ **do**
      $\mathcal{Y} = \mathcal{X} \times_1 U_1^\intercal \times_2 \cdots \times_{k-1} U_{k-1}^\intercal \times_{k+1} U_{k+1}^\intercal \times_{k+2} \cdots \times_d U_d^\intercal$
      $U_k \leftarrow \text{LLSV}(Y_{(k)}, r_k)$
    end for
  end for
  $\mathcal{G} \leftarrow \mathcal{Y} \times_d U_d^\intercal$ ▷ update core
  return $[\mathcal{G}, U_{1:d}]$ ▷ $\mathcal{T} \equiv \{\mathcal{G}; U_{1:d}\}$
end function

HOOI is a good algorithm, but it can be better. As previously mentioned, we introduce three optimizations for the HOOI algorithm in an attempt to make it more competitive against STHOSVD.

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 STHOSVD. We can reduce the cost of TTMs by avoiding redundant computations. Notice that for $k = 1$ in Algorithm 6 the following multi-TTM is computed $\mathcal{Y} = \mathcal{X} \times_2 \mathbf{U}_2^\intercal \times_3 \mathbf{U}_3^\intercal \cdots \times_d \mathbf{U}_d$. At $k = 2$ the multi-TTM is $\mathcal{Y} = \mathcal{X} \times_2 \mathbf{U}_1 \times \times_3 \mathbf{U}_3^\intercal \cdots \times_d \mathbf{U}_d$. 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 [7] and has been applied to Tucker computations as well [8, 9]. 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 7 shows the HOOI iteration using dimension tree memoization implemented recursively.

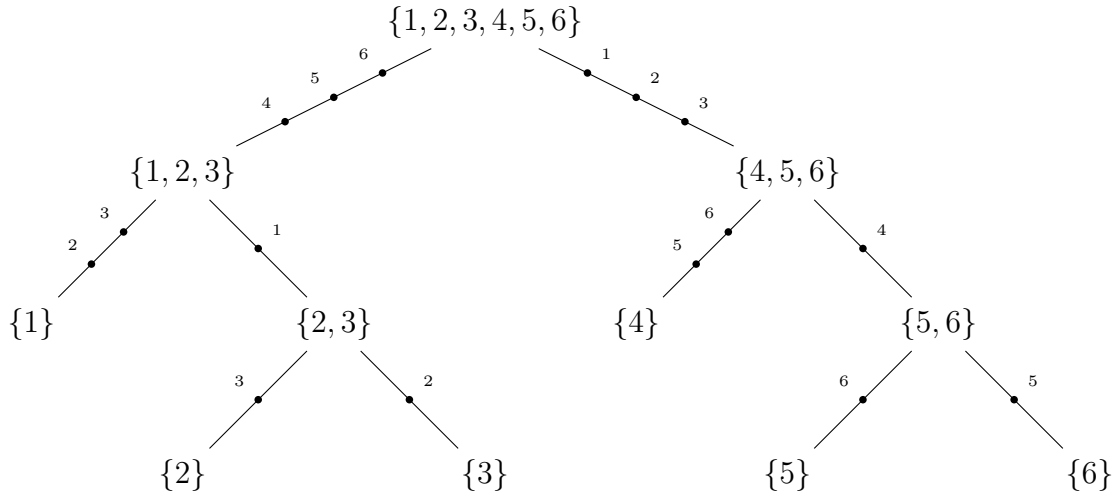


Figure 3.2: Illustration of multi-TTM memoization for an order-6 tensor. Each node in the tree shows the set of modes in which multiplication has not been performed. Each notch in an edge is a TTM in the labeled mode. Factor matrices are computed at each leaf node in the mode shown. $\mathcal{G}$ is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

Algorithm 7 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**
    $\mathbf{U}_m = \text{LLSV}(\mathbf{X}_{(m)}, \mathbf{U}_m, r_m)$
    if $m = d$ **then**
      $\mathcal{G} = \mathcal{X} \times_d \mathbf{U}_m^\intercal$
    end if
  else
    Partition $\mathbf{m} = [\mu, \eta]$
    $\mathcal{X} = \mathcal{X} \bigtimes_{k \in \mu} \mathbf{U}_k^\intercal$
    $[\mathcal{G}, \{\mathbf{U}_k\}] = \text{HOSI-DT}(\mathcal{X}, \{\mathbf{U}_k\}, \eta, \mathbf{r})$
    $\mathcal{X} = \mathcal{X} \bigtimes_{k \in \eta} \mathbf{U}_k^\intercal$
    $[\mathcal{G}, \{\mathbf{U}_k\}] = \text{HOSI-DT}(\mathcal{X}, \{\mathbf{U}_k\}, \mu, \mathbf{r})$
  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 iterations. Algorithm 8 shows a single subspace iteration, but in principle, the computations could be repeated to improve accuracy.

Algorithm 8 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 6 or $\mathbf{X}_{(m)}$ from Algorithm 7, 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 8 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

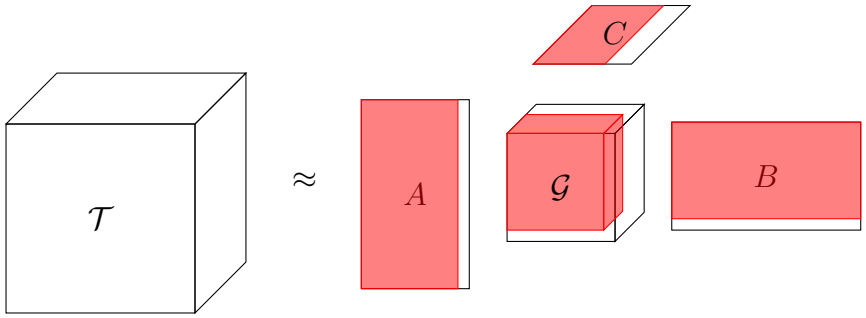


Figure 3.3: Adaptive HOOI

with column 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 STHOSVD 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 $\hat{\mathbf{X}} = [\mathcal{G}; \mathbf{U}_1, \ldots, \mathbf{U}_d]$ for a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ such that $\|\hat{\mathbf{X}} - \mathcal{X}\| \leq \varepsilon \|\mathcal{X}\|$. 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 clasical HOOI the core is only updated after the iterations, here we compute the core tensor at the end of everu 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 $\|\mathcal{X} - \hat{\mathbf{X}}\|^2 = \|\mathcal{X} - \mathcal{G} \times_1 \mathbf{U}_1 \times \cdots \times_d \mathbf{U}_d\|^2 = \|\mathcal{X}\|^2 - \|\mathcal{G}\|^2$ ([3, 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

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

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 closest index satisfying (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 small factor α before trying again. Typically, $\alpha \approx 2$ is sufficient. The details of this algorithm, are described in 9. In practice, we do the optimization problem above in a way that minimizes the memory footprint. We do so by exploiting HOOI's form of the immediate core $\mathcal{G}$. We use the immediate form to compute its cumulative sum of the squared core $\mathcal{G}^2$ (square all entries in the core) and then consider all values of this cumulative sum squared tensor to find the first instance that satisfies the error tolerance to get the new ranks.

Algorithm 9 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 STHOSVD

STHOSVD'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 STHOSVD flops cost in Table 3.1 (shown in red).

STHOSVD'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

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

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 STHOSVD (as we observe in our experiments). We summarize the STHOSVD 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

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

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 $\mathbf{Y}_{(k)}\mathbf{Y}'_{(k)}$ 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 $2^{nr^d}/_P$ 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 STHOSVD, 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 $\mathcal{G} = \mathcal{Y} \times_k \mathbf{U}_{(k)}^\mathsf{T}$ (in the notation of Algorithm 6) and the second computes the tensor contraction $\mathbf{Y}_{(k)}\mathbf{G}_{(k)}^\mathsf{T}$. 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 Eq. (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

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**	$4\frac{rn^d}{P}$	
STHOSVD	$\frac{n^{d+1}}{P} + \mathcal{O}(dn^3)$		$2\frac{rn^d}{P}$		-
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 STHOSVD and HOOI with Subspace Iteration and Dimension Trees (HOSI-DT) optimizations. We assume ℓ iterations of HOSI-DT are performed.

be large for HOOI to improve performance over STHOSVD, 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.	$\frac{rn^{d-1}}{P}(P_1-1)+\frac{rn^{d-1}}{P}(P_d-1)$	
STHOSVD	$\frac{n^d}{P}\frac{P_1-1}{P_1}+dn^2$		$\frac{rn^{d-1}}{P}(P_1-1)$		-
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 STHOSVD 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 [10].

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 [11] (3-way), HCCI [12] (4-way), and SP [13] (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

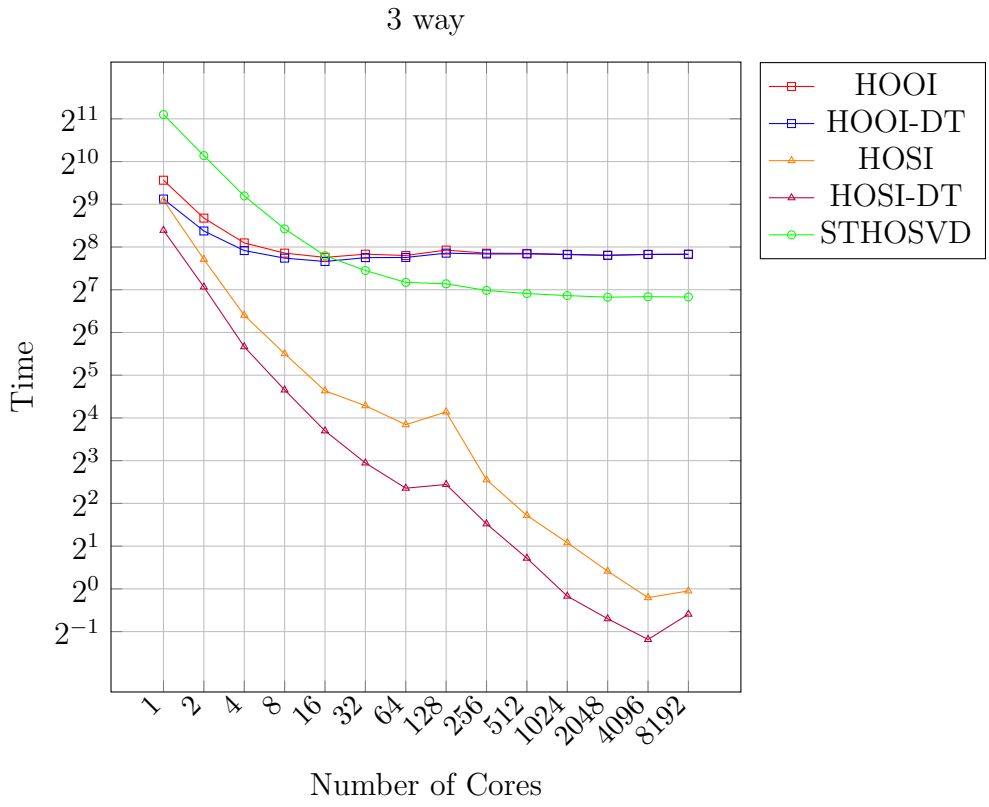


Figure 3.4: Strong scaling comparison of Tucker algorithms in single precision using a 3-way $\mathbf{3750} \times \mathbf{3750} \times \mathbf{3750}$ input tensor

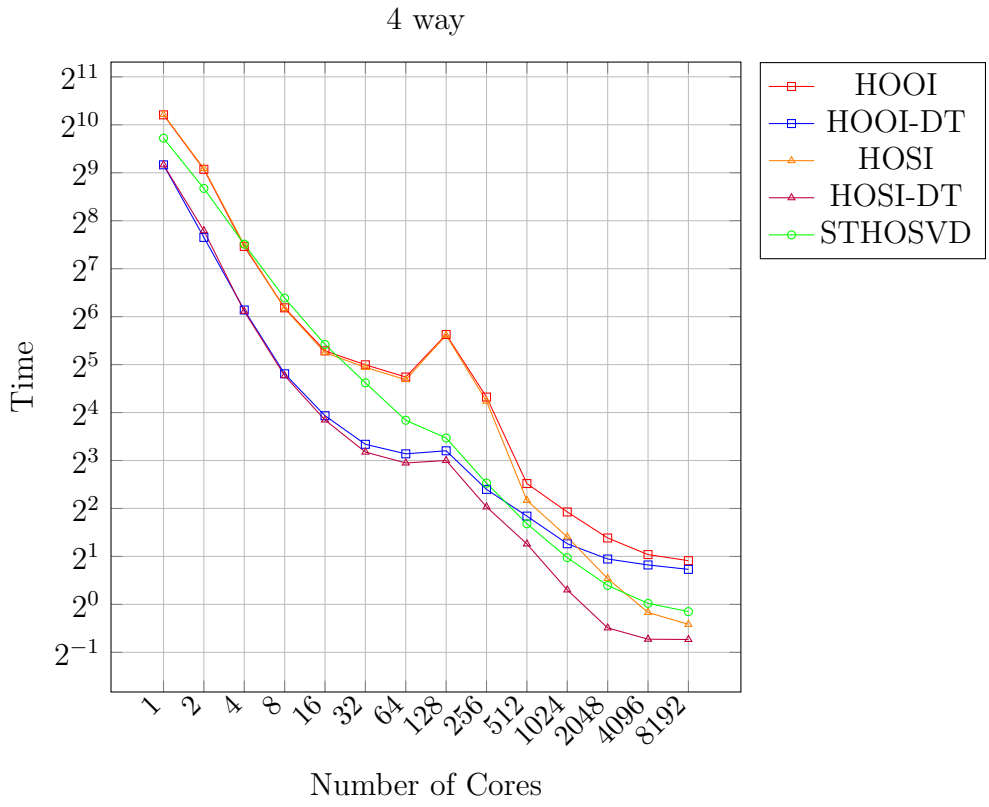


Figure 3.5: Strong scaling comparison of Tucker algorithms in single precision using a 4-way $\mathbf{560} \times \mathbf{560} \times \mathbf{560} \times \mathbf{560}$ input tensor

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 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 Fig. 3.6, 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 Figure 3.6. 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 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

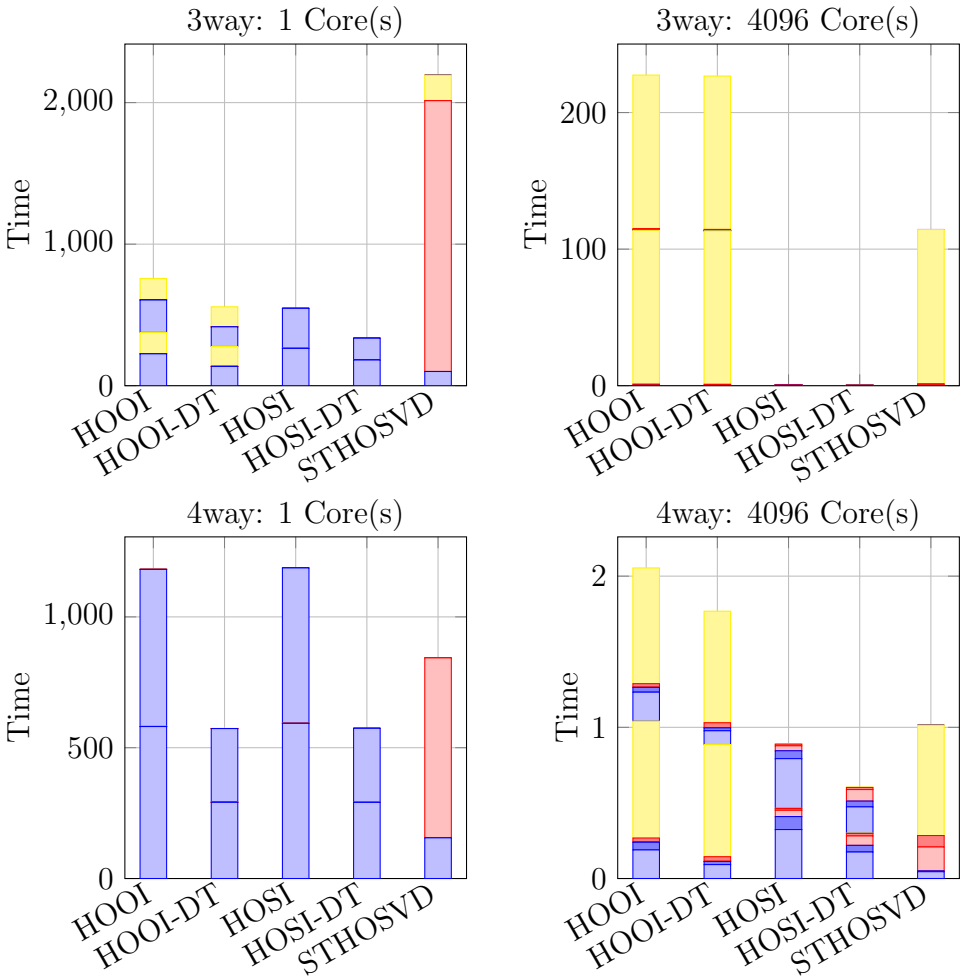


Figure 3.6: Running time breakdown for the synthetic 3-way (top) and 4-way (bottom) tensors

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 bottleneckm 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 computation. 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 $\frac{n}{r}$.

Overall There are two factors that are pertinent to both 3-way and 4-way experiments. The first is the spike noticible at 128 for most (some more than others) of the algorithms. These come from a choice of configuration of the slurm jobs regarding the binding of NUMA regions to the cores. There were two main options available to run these experiments with, *cores* and *map_ldom*. The former assigns all the NUMA regions of the node to the running cores, and the latter we most perform a manual assignment of the NUMA regions which gives us more control. We are not utilizing the full potential of a node from 1 to 64 cores, as there are more cores available, but the *cores* configuration still assigns all the NUMA regions of the node to the running cores, so these cores have more memory than what is 'pertinent' to them. Thus, when we first use the full potential of a node at 128 cores, there is now a competition for resources. Running the *map_ldom* configuration allows us to bypass that limitation,

so from 1 to 64 cores we only assign NUMA regions that would be 'pertinent' to those cores assuring a steady parallel scaling in terms of memory. This makes the experiments from 1 to 32 nodes equally slower to all five algorithms. In other words it removes the spike, simply because the rate of increase is steady now. The other factor regards why HOSI and HOSI-DT stop scaling at 8192 cores, the reason is 42.

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 Fig. 3.10 it can be seen that the HOSI-DT (Over) 0.1 threshold reached the desired at the first iteration, so we don't show

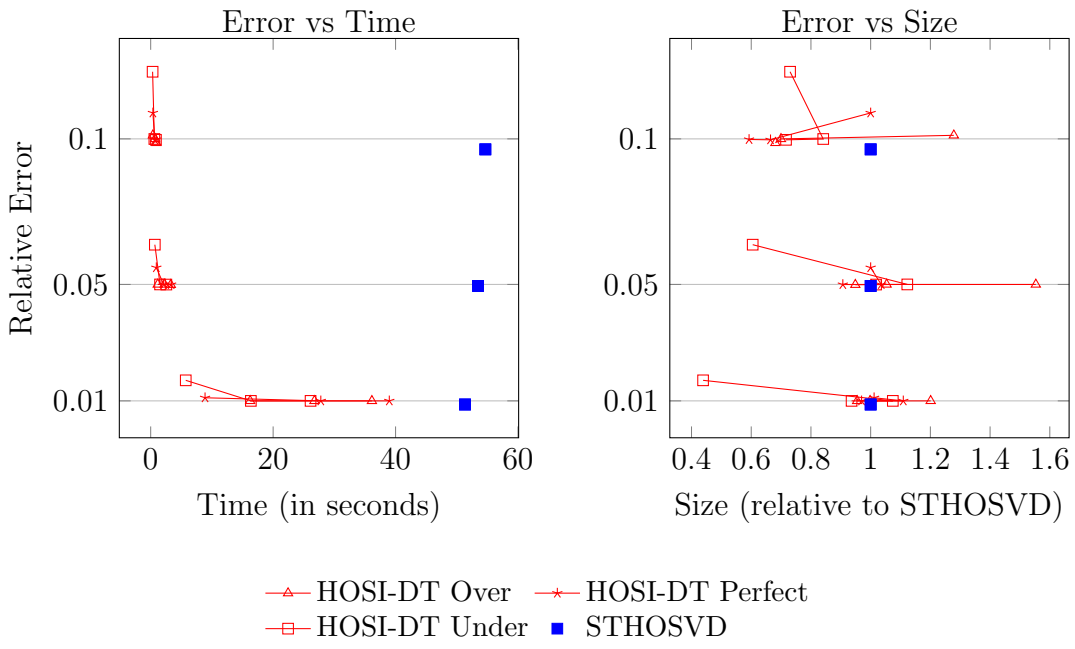


Figure 3.7: Progression of time, error, and relative size over 3 iterations of rank-adaptive HOSI-DT on the Miranda dataset using 1024 cores.

the breakdown for the second iteration despite its total time being shown on Fig. 3.9.

Miranda (3-way)

The Miranda dataset is a three-dimensional simulation data of the density ratios of non-reacting flow of viscous fluids [11]. 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 3.7 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

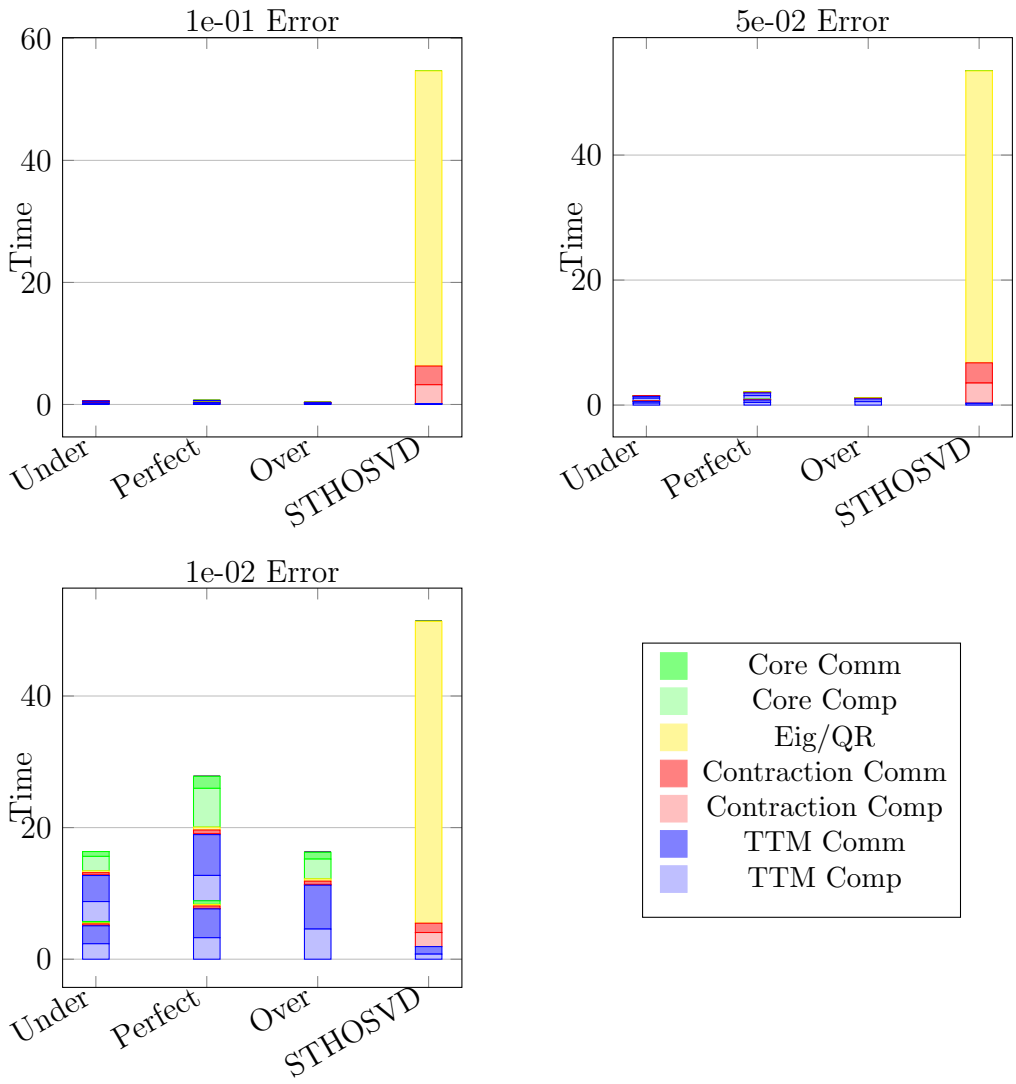


Figure 3.8: Running time breakdown for the Miranda dataset using 1024 cores under different levels of compression.

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-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 [12]. The dimension 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 [13]. 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 3.9 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 3.10 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 3.11 shows that we can typically obtain better compression after three iterations. For example, overestimating the ranks for low compression yields a speedup 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, Fig. 3.12 shows that when starting from perfect estimates of the ranks for mid compression, HOSI-DT gets the desired error

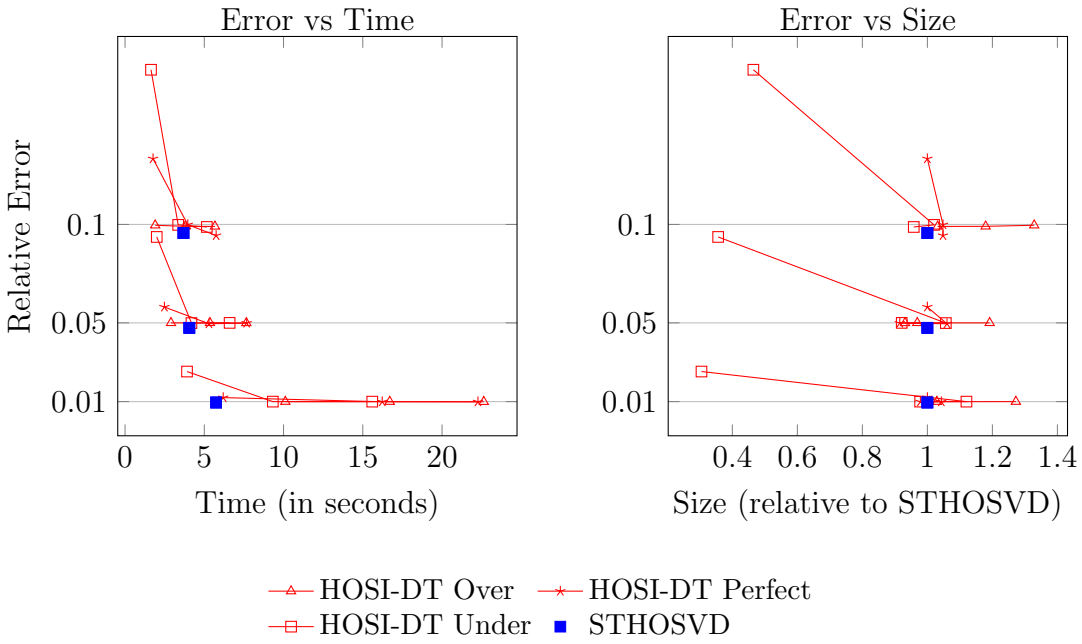


Figure 3.9: Progression of time, error, and relative size over 3 iterations of rank-adaptive HOSI-DT on the HCCI dataset using 128 cores.

tolerance and same compression ratio in less time than STHOSVD, with HOSI-DT achieving a 1.4× speedup.

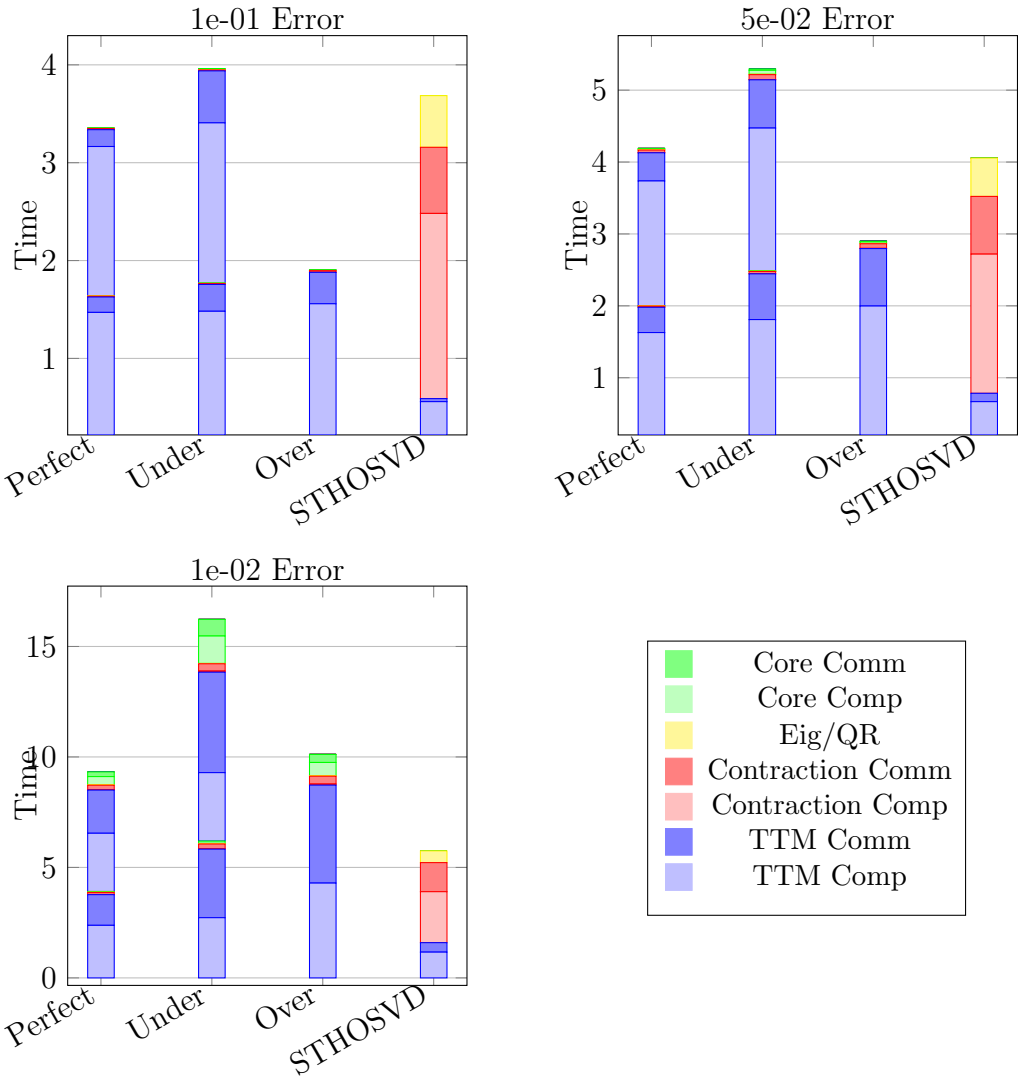


Figure 3.10: Running time breakdown for the HCCI dataset using 128 cores under different levels of compression.

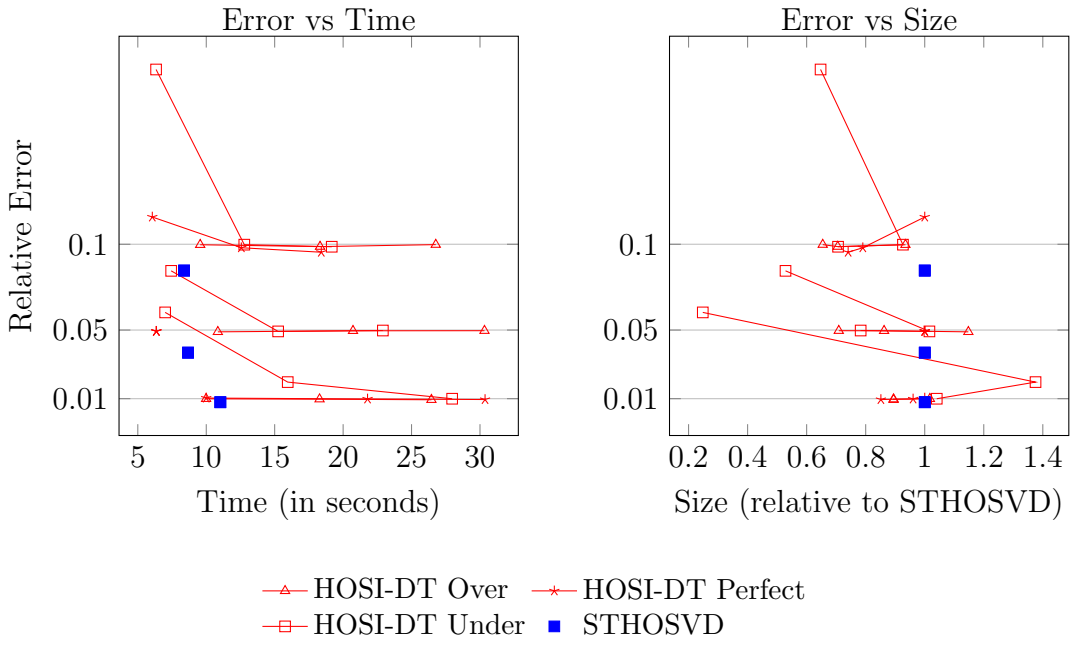


Figure 3.11: Progression of time, error, and relative size over 3 iterations of rank-adaptive HOSI-DT on the SP dataset using 2048 cores.

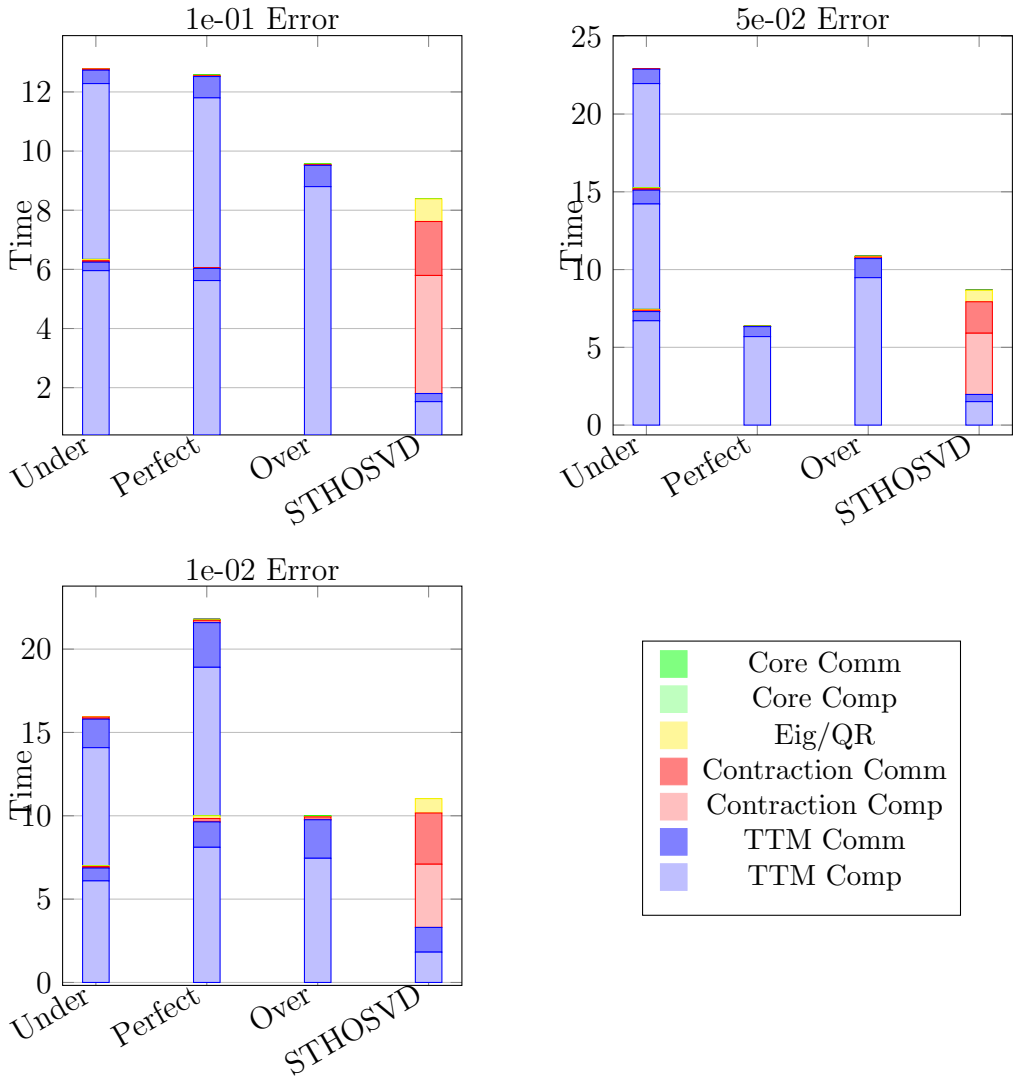


Figure 3.12: Running time breakdown for the SP dataset using 2048 cores under different levels of compression.

Bibliography

[1] Thomas Lam. *100 Questions: A Mathematical Conventions Survey*. Accessed: 2024-09-09. 2024. URL: https://cims.nyu.edu/~tjl8195/survey/results.html.

[2] Tamara G. Kolda and Grey Ballard. *Tensor Decompositions for Data Science*. Cambridge University Press, 2024.

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

[4] 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.

[5] 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/S0895479989834699.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.