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

To all of those that came before me, and to all of those who are yet to come

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF ABREVIATIONS

Number sets

$\mathbb{C}$ Complex numbers

$\mathbb{H}$ Quaternions

$\mathbb{R}$ Real numbers

Other symbols

ρ Friction index

V Constant volume

Physics constants

c Speed of light in a vacuum

G Gravitational constant

h Planck constant

ABSTRACT

This thesis is built on two projects. They all fall under Tensor Decompositions as the title states. Here I present 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 implemen-

tation 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 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 title of this first section is named after question 45 of that survey, *What is a Tensor?*. Its results 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 *"94% 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.

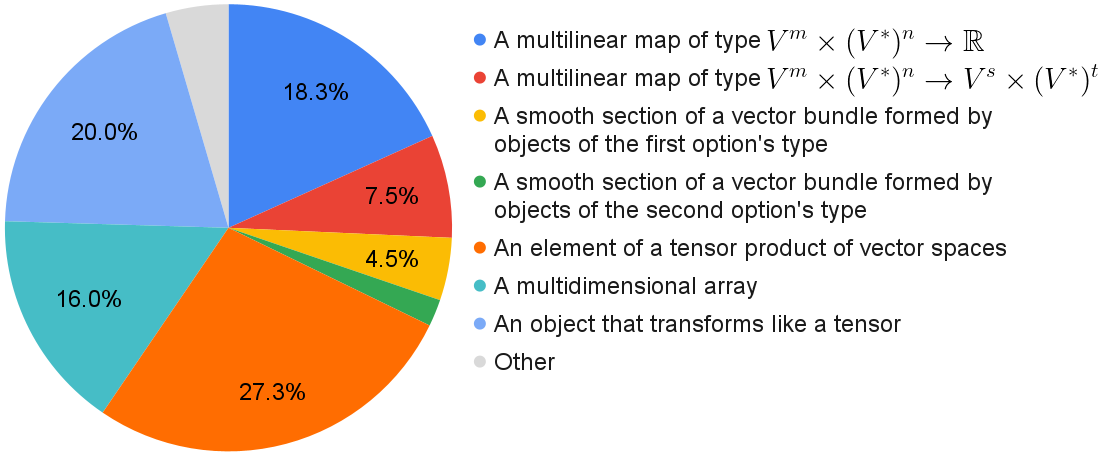

Figure 1.1: What Is A Tensor?

1.2 Problem Statement

1.2.1 Preliminaries

A tensor is a multidimensional array. Think of an array as an 1D tensor, a matrix as a 2D tensor, a 3D tensor is a cube of data, a 4D tensor is an array of 3D tensors, a 5D a matrix of 3D tensors, and it generalizes onwards. Similarly, a scalar can be thought of as a 0D tensor. The number of entries stored in memory is the product of each dimension size. A 1D tensor (an array) has n entries, a 2D Tensor (a matrix) has n^2 entries, a 3D tensor has n^3 and so on. As you can imagine, the memory footprint of a tensor gets larger and larger the more dimensions and the bigger their sizes are. Because they get memory expensive so quickly, there are ways to compress these multidimensional arrays that go beyond the regular zipping of a file. *consider*

motivating why we prefer this over regular file compression. We call the compression of a tensor to be a **tensor decomposition** because it relies on decomposing the full tensor into smaller subpart that when multiplied together we get an approximation of the original tensor. Though there are only a couple of types of tensor decompositions, each type of decomposition has many algorithms of which we will only focus on a few on this thesis. We will only cover two types of tensor decompositions, one belonging to each project:

Tucker Tensor

This is the type of tensor used on project 2. A Tucker Tensor (TTensor) is a core tensor with as many factor matrices as there are dimensions which are multiplied with the core to reconstruct the global tensor as in Figure 1.2.

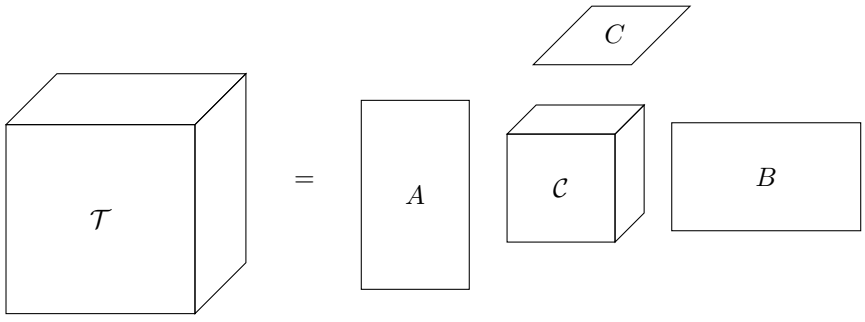

Figure 1.2: A 3 way Tucker Tensor Diagram

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 approximation gets more accurate with a

bigger core tensor, which implies a trade-off between compression and accuracy. The traditional methods are HOSVD (Higher Order Singular Value Decomposition) and STHOSVD (Sequentially Truncated Singular Value Decomposition). HOOI (Higher Order Orthogonal Iteration) was the last to be added to this list, this is the method project 2 builds on. STHOSVD and HOOI can be seen in Algorithms 1 and 2.

Kruskal Tensor

This tensor decomposition format is used on project 1. A Kruskal Tensor (KTensor) is a number of d outer products of vectors of length n, the number of outer products is the rank of the KTensor namely R, which can be seen in Figure 1.3. The memory

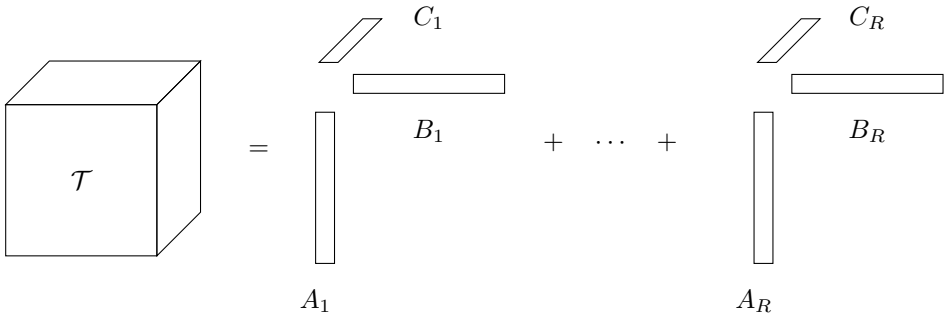

Figure 1.3: A 3 way CP Tensor Diagram

footprint of a K Tensor is $3Rn$. The approximation gets more accurate as R increases. This is the 2D the matrix equivalent of having a matrix approximation to be the outer product of two vectors as an $n \times 1$ array times a $1 \times n$ array equals an $n \times n$ matrix. The traditional methods are gradient descent and the newton method. The process of compressing a d-way tensor into a K Tensor is called a CP Decomposition, there

are several CP Decomposition methods, but we focus on the damped gauss newton (CP-DGN) optimization method which can be found in Algorithm 5.

Tensor Train

This tensor decomposition format is used on project 3. A Tensor Train Tensor (TT Tensor) takes a d-way tensor and compresses it into a product of two outer matrices and $d-2$ 3way cores. The memory footprint of a TT tensor is $2nr + dnr^2$, and it is

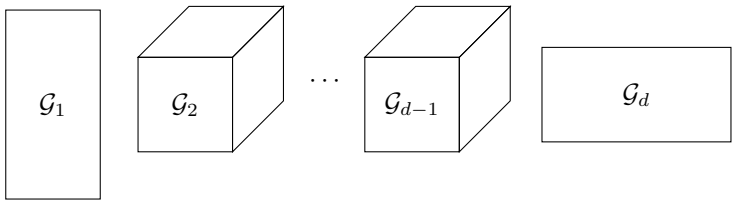

Figure 1.4: A d way Tensor Train Diagram

quite effective for tensor with multiple dimensions. The traditional method for TT is the TT-SVD.

1.2.2 The Matrix Multiplication Tensor

There is a specific type of tensor that will be necessary for the understanding of the first project; the Matrix Multiplication Tensor. We assume the reader is already familiar with matrix matrix multiplication, which makes its tensor format easier to comprehend. It is a 3D tensor which holds the matrix multiplication algorithm. We can create this tensor for any two matrices $m \times n$ times an $n \times p$. The dimension of

the matrix multiplication tensor would then be $m^2 \times n^2 \times p^2$. To successfully multiply two matrices using the tensor, we vectorize the two matrices and multiply against the first and second dimensions of the tensor, respectively, to get the output you normally get in matrix multiplication, as shown on Figure 1.5.

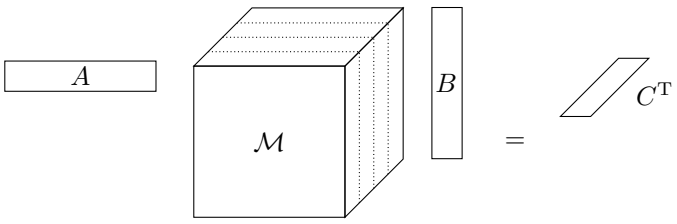

Figure 1.5: Matrix Multiplication in Tensor Format

1.2.3 The Importance of Tensor Decompositions

Tensor Decompositions for Data Compression ($2^{\mathbf{nd}}$ and $3^{\mathbf{rd}}$ projects)

As discussed earlier, the memory footprint increases quickly as the number of dimensions increases as well as its sizes. Since the size of a tensor is its dimensions multiplied, an increased in just a single of its dimensions can result in an extremely larger tensor. In fact the TT tensor is an effective option for tensors with many dimensions as the storage is exponential in the number of dimensions (recall storage cost is n^d for a uniform tensor). This happens because the Tucker Tensor still has exponential in d storage r^d which the TT tensor does not.

Because of the high cost of storing raw tensors, tensor decompositions are highly effective in reducing tensor storage. However, we cannot decrease memory footprints

without first losing some accuracy. Classic HOOI, STHOSVD, and the TTSVD algorithms allow the user to specify the dimensions of the core tensor. Often times just specifying the dimensions does not mean the user will know how much error it will introduce to the tensor. This is not a problem for STHOSVD and TTSVD since those have an error-specified variant that the HOOI algorithm does not.

Tensor Decompositions for Interpretations (1st project)

Recall that matrix multiplication is an $O(n^3)$ algorithm. But we can decrease this cost by using Strassen's Algorithm as shown below:

Classic Algorithm

$$
\begin{aligned}
M_1 &= A_{11} \cdot B_{11} \\
M_2 &= A_{12} \cdot B_{21} \\
M_3 &= A_{11} \cdot B_{12} \\
M_4 &= A_{12} \cdot B_{22} \\
M_5 &= A_{21} \cdot B_{11} \\
M_6 &= A_{22} \cdot B_{21} \\
M_7 &= A_{21} \cdot B_{12} \\
M_8 &= A_{22} \cdot B_{22} \\
C_{11} &= M_1 + M_2 \\
C_{12} &= M_3 + M_4 \\
C_{21} &= M_5 + M_6 \\
C_{22} &= M_7 + M_8
\end{aligned}
$$

8 multiplies, 4 additions

$$T(n) = 8T(n/2) + O(n^2)$$

$$T(n) = O(n^{log_2 8}) = O(n^3)$$

Strassen's Algorithm 7 multiplies, 18 additions

$$M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \qquad T(n) = 7T(n/2) + O(n^2)$$

$$M_2 = (A_{12} + A_{22}) \cdot B_{11} \qquad T(n) = O(n^{log_2 7}) = O(n^{2.81})$$

$$M_3 = A_{11} \cdot (B_{21} - B_{22})$$

$$M_4 = A_{22} \cdot (B_{12} - B_{11})$$

$$M_5 = (A_{11} + A_{21}) \cdot B_{22}$$

$$M_6 = (A_{12} - A_{11}) \cdot (B_{11} + B_{21})$$

$$M_7 = (A_{21} - A_{22}) \cdot (B_{12} + B_{22})$$

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

This is a product of the recursive implementation of Strassen's algorithm, the less recursive calls we do, the less expensive our algorithm will be as shown by the cost analysis above. For the 2×2 case we know we cannot improve the 7 multiplies from *insert reference here*. Furthermore, we also know that Strassen's algorithm is not unique. We can rearrange the additions and multiplies to get another algorithm with the same number of multiplies and additions. We could search for different solutions that all have the same cost by performing an optimization search on each parameter

$$
\begin{aligned}
M_1 &= \left(u_{11}^{(1)} A_{11} + u_{12}^{(1)} A_{12} + u_{21}^{(1)} A_{21} + u_{22}^{(1)} A_{22}\right) \cdot \left(v_{11}^{(1)} B_{11} + v_{12}^{(1)} B_{12} + v_{21}^{(1)} B_{21} + v_{22}^{(1)} B_{22}\right) \\
M_2 &= \left(u_{11}^{(2)} A_{11} + u_{12}^{(2)} A_{12} + u_{21}^{(2)} A_{21} + u_{22}^{(2)} A_{22}\right) \cdot \left(v_{11}^{(2)} B_{11} + v_{12}^{(2)} B_{12} + v_{21}^{(2)} B_{21} + v_{22}^{(2)} B_{22}\right) \\
M_3 &= \left(u_{11}^{(3)} A_{11} + u_{12}^{(3)} A_{12} + u_{21}^{(3)} A_{21} + u_{22}^{(3)} A_{22}\right) \cdot \left(v_{11}^{(3)} B_{11} + v_{12}^{(3)} B_{12} + v_{21}^{(3)} B_{21} + v_{22}^{(3)} B_{22}\right) \\
M_4 &= \left(u_{11}^{(4)} A_{11} + u_{12}^{(4)} A_{12} + u_{21}^{(4)} A_{21} + u_{22}^{(4)} A_{22}\right) \cdot \left(v_{11}^{(4)} B_{11} + v_{12}^{(4)} B_{12} + v_{21}^{(4)} B_{21} + v_{22}^{(4)} B_{22}\right) \\
M_5 &= \left(u_{11}^{(5)} A_{11} + u_{12}^{(5)} A_{12} + u_{21}^{(5)} A_{21} + u_{22}^{(5)} A_{22}\right) \cdot \left(v_{11}^{(5)} B_{11} + v_{12}^{(5)} B_{12} + v_{21}^{(5)} B_{21} + v_{22}^{(5)} B_{22}\right) \\
M_6 &= \left(u_{11}^{(6)} A_{11} + u_{12}^{(6)} A_{12} + u_{21}^{(6)} A_{21} + u_{22}^{(6)} A_{22}\right) \cdot \left(v_{11}^{(6)} B_{11} + v_{12}^{(6)} B_{12} + v_{21}^{(6)} B_{21} + v_{22}^{(6)} B_{22}\right) \\
M_7 &= \left(u_{11}^{(7)} A_{11} + u_{12}^{(7)} A_{12} + u_{21}^{(7)} A_{21} + u_{22}^{(7)} A_{22}\right) \cdot \left(v_{11}^{(7)} B_{11} + v_{12}^{(7)} B_{12} + v_{21}^{(7)} B_{21} + v_{22}^{(7)} B_{22}\right)
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= w_{11}^{(1)} M_1 + w_{11}^{(2)} M_2 + w_{11}^{(3)} M_3 + w_{11}^{(4)} M_4 + w_{11}^{(5)} M_5 + w_{11}^{(6)} M_6 + w_{11}^{(7)} M_7 \\
C_{12} &= w_{12}^{(1)} M_1 + w_{12}^{(2)} M_2 + w_{12}^{(3)} M_3 + w_{12}^{(4)} M_4 + w_{12}^{(5)} M_5 + w_{12}^{(6)} M_6 + w_{12}^{(7)} M_7 \\
C_{21} &= w_{21}^{(1)} M_1 + w_{21}^{(2)} M_2 + w_{21}^{(3)} M_3 + w_{21}^{(4)} M_4 + w_{21}^{(5)} M_5 + w_{21}^{(6)} M_6 + w_{21}^{(7)} M_7 \\
C_{22} &= w_{22}^{(1)} M_1 + w_{22}^{(2)} M_2 + w_{22}^{(3)} M_3 + w_{22}^{(4)} M_4 + w_{22}^{(5)} M_5 + w_{22}^{(6)} M_6 + w_{22}^{(7)} M_7
\end{aligned}
$$

Figure 1.6: The optimization search for Matrix Multiply Algorithms for the 2×2 case involving 7 multiplications. The number of variables is $4 \cdot 7 \cdot 3 = 84$. If we are searching for a discrete solution with only $-1, 0, 1$ as coefficients then we have $3^{84} \approx 10^{40}$ possibilities

involved in matrix multiplication as seen in Figure 1.6:

Below is one of these Strassen's variant algorithm next to the original. Strassen's algorithm has an underlying structure that this project relies on. However, it is not visible on the original layout. Strassen's algorithm is **Cyclic Invariant**. This is hard to visualize with only this presentation. Instead, we must turn to tensors to understand this structure.

Strassen's Algorithm	**Variant of Strassen's Algorithm**
$M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$	$M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
$M_2 = (A_{12} + A_{22}) \cdot B_{11}$	$M_2 = (A_{12} + A_{22}) \cdot B_{11}$
$M_3 = A_{11} \cdot (B_{21} - B_{22})$	$M_3 = A_{22} \cdot (B_{12} - B_{11})$
$M_4 = A_{22} \cdot (B_{12} - B_{11})$	$M_4 = (A_{21} - A_{22}) \cdot (B_{12} + B_{22})$
$M_5 = (A_{11} + A_{21}) \cdot B_{22}$	$M_5 = (A_{11} + A_{21}) \cdot B_{22}$
$M_6 = (A_{12} - A_{11}) \cdot (B_{11} + B_{21})$	$M_6 = A_{11} \cdot (B_{21} - B_{22})$
$M_7 = (A_{21} - A_{22}) \cdot (B_{12} + B_{22})$	$M_7 = (A_{12} - A_{11}) \cdot (B_{11} + B_{21})$
$C_{11} = M_1 + M_4 - M_5 + M_7$	$C_{11} = M_1 + M_3 - M_4 - M_6$
$C_{12} = M_3 + M_5$	$C_{12} = M_2 + M_3$
$C_{21} = M_2 + M_4$	$C_{21} = M_5 + M_6$
$C_{22} = M_1 - M_2 + M_3 + M_6$	$C_{22} = M_1 - M_2 + M_5 + M_7$

Recall that for a CP-Tensor, its rank is the number of components, but they can also be viewed as the number of columns of the factor matrices. When we decompose the matrix multiplication tensor we can find matrix multiplications algorithms embedded into its CP decomposition! Furthermore, the rank of the CP decomposition corresponds to the number of multiplications in the algorithm. Therefore, we can actively search for matrix multiplication algorithms with specified number of multi-

plications by decomposing the MatMul tensor. The CP decomposition that contains both Strassen's Algorithm and its variant is shown below. The three blocks represent the three factor matrices, and the columns are the components of the 'chicken feet' in Figure 3. '

	M_1	M_2	M_3	M_4	M_5	M_6	M_7
A_{11}	1	0	1	0	1	−1	0
A_{12}	0	1	0	0	0	1	0
A_{21}	0	0	0	0	1	0	1
A_{22}	1	1	0	1	0	0	−1
B_{11}	1	1	0	−1	0	1	0
B_{12}	0	0	0	1	0	0	1
B_{21}	0	0	1	0	0	1	0
B_{22}	1	0	−1	0	1	0	1
C_{11}	1	0	0	1	−1	0	1
C_{21}	0	0	1	0	1	0	0
C_{12}	0	1	0	1	0	0	0
C_{22}	1	−1	1	0	0	1	0

	M_1	M_2	M_3	M_4	M_5	M_6	M_7
A_{11}	1	0	0	0	1	1	−1
A_{12}	0	1	0	0	0	0	1
A_{21}	0	0	0	1	1	0	0
A_{22}	1	1	1	−1	0	0	0
B_{11}	1	1	−1	0	0	0	1
B_{12}	0	0	1	1	0	0	0
B_{21}	0	0	0	0	0	1	1
B_{22}	1	0	0	1	1	−1	0
C_{11}	1	0	1	1	−1	0	0
C_{21}	0	0	0	0	1	1	0
C_{12}	0	1	1	0	0	0	0
C_{22}	1	−1	0	0	0	1	1

By simply rearranging the columns of the original algorithm, the cyclic structure the algorithm can be seen in the colors of the decomposition. Notice how we have repeats of the same subcomponents in a cyclic form, they are called the **cyclic components** of the decomposition. That is we have green-purple-orange, then orange-green-purple and lastly purple-orange-green. The red color up front does not cycle, but is repeated in all three factor matrices, that is called the **symmetric component**. It is called that way because if we reshape that vector into a 2×2 matrix, it becomes a symmetric matrix. We can reduce the number of variables we must search from the 3^{84} possibilities to just 3^{28} possibilities. This however does not mean

that exhausting the cyclic invariant solutions implies exhausting all solutions. As the original Strassen algorithm shows us, not all solutions are cyclic invariant. Now we can change the number of columns that are symmetric which would then change the number of columns that are cyclic. As long as we have that number of columns is equal to the number of columns in the symmetric component plus 3 times the number of columns in the cyclic component. In other words: $R = Rs + 3Rc$:

	M_1	M_2	M_3	M_4	M_5	M_6	M_7
A_{11}	1	0	0	0	1	1	-1
A_{12}	0	1	0	0	0	0	1
A_{21}	0	0	0	1	1	0	0
A_{22}	1	1	1	-1	0	0	0
B_{11}	1	1	-1	0	0	0	1
B_{12}	0	0	1	1	0	0	0
B_{21}	0	0	0	0	0	1	1
B_{22}	1	0	0	1	1	-1	0
C_{11}	1	0	1	1	-1	0	0
C_{21}	0	0	0	0	1	1	0
C_{12}	0	1	1	0	0	0	0
C_{22}	1	-1	0	0	0	1	1

	M_1	M_2	M_3	M_4	M_5	M_6	M_7
A_{11}	1	0	0	0	0	1	0
A_{12}	0	0	-1	1	0	1	-1
A_{21}	0	1	0	-1	-1	-1	0
A_{22}	0	1	1	-1	0	-1	0
B_{11}	1	0	0	0	0	0	1
B_{12}	0	0	-1	1	-1	0	1
B_{21}	0	1	0	-1	0	-1	-1
B_{22}	0	1	1	-1	0	0	-1
C_{11}	1	0	0	0	1	0	0
C_{21}	0	0	-1	1	1	-1	0
C_{12}	0	1	0	-1	-1	0	-1
C_{22}	0	1	1	-1	-1	0	0

These two algorithms differ in the number of symmetric and cyclic components. At first glance one may think that besides that difference, they are the same "Strassen" algorithm. However, that is not the case. If we look at the algorithmic version of the

CP decomposition above we will find that though they both have 7 multiplication, the one with one symmetric component has 18 additions while the one with four symmetric components has 24 additions, thus being suboptimal.

Variant of Strassen's Algorithm (Rs=1 Rc=2)

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_2 &= (A_{12} + A_{22}) \cdot B_{11} \\
M_3 &= A_{22} \cdot (B_{12} - B_{11}) \\
M_4 &= (A_{21} - A_{22}) \cdot (B_{12} + B_{22}) \\
M_5 &= (A_{11} + A_{21}) \cdot B_{22} \\
M_6 &= A_{11} \cdot (B_{21} - B_{22}) \\
M_7 &= (A_{12} - A_{11}) \cdot (B_{11} + B_{21}) \\
C_{11} &= M_1 + M_3 - M_4 - M_6 \\
C_{12} &= M_2 + M_3 \\
C_{21} &= M_5 + M_6 \\
C_{22} &= M_1 - M_2 + M_5 + M_7
\end{aligned}
$$

Variant of Strassen's Algorithm (Rs=4 Rc=1)

$$
\begin{aligned}
M_1 &= A_{11} \cdot B_{11} \\
M_2 &= (A_{12} + A_{22}) \cdot (B_{12} + B_{22}) \\
M_3 &= (A_{22} - A_{21}) \cdot (B_{22} - B_{21}) \\
M_4 &= (A_{21} - A_{12} - A_{22}) \cdot (B_{21} - B_{12} - B_{22}) \\
M_5 &= (-A_{12}) \cdot (-B_{21}) \\
M_6 &= (A_{11} - A_{12} + A_{21} - A_{22}) \cdot (-B_{12}) \\
M_7 &= (-A_{21}) \cdot (B_{11} - B_{12} + B_{21} - B_{22}) \\
C_{11} &= M_1 + M_5 \\
C_{22} &= M_4 - M_3 + M_5 + M_6 \\
C_{22} &= M_2 - M_4 - M_5 + M_7 \\
C_{22} &= M_2 + M_3 - M_4 + M_5
\end{aligned}
$$

The CP Decomposition

The Tucker Decomposition

3.1 Literature Review

Much of the work presented in this thesis proposal comes mainly from Kolda and Ballard's soon to be released book [2]. In fact, little was taken from outside it. But here are some other meaningful references [3], [4], [5], [6], [7], [8], [9], [10], [11]

3.2 Proposed Approach

For the first and second projects, we are simply implementing the algorithms in the TuckerMPI library mentioned earlier. For the second project, however, there is an additional contribution. We modify classic HOOI to have both rank and error specified versions like STHOSVD. To understand this process, consider the optimal rank-$\mathbf{R}$ Tucker Decomposition of a tensor $\mathcal{X}$ can be expressed as a solution to the optimization problem

$$\min||\mathcal{X} - \mathcal{G} \times U_1 \times_2 \cdots \times_d U_d||$$

$$\text{subject to } \mathcal{G} \in \mathbb{R}^{n_1 \times \cdots \times n_d}, U_k \in \mathbb{R}^{n_k \times r_k} \ \forall k \in [d]$$

Now suppose we have a relative error tolerance of how accurate we want our approximation to be. Then the approximation must satisfy:

$$\frac{||\mathcal{X} - \mathcal{T}||}{||\mathcal{X}||} \leq \epsilon$$

$$\frac{||\mathcal{X} - \mathcal{T}||^2}{||\mathcal{X}||^2} \leq \epsilon^2$$

$$\therefore ||\mathcal{X} - \mathcal{T}||^2 \leq \epsilon^2 \cdot ||\mathcal{X}||^2$$

$$\therefore ||\mathcal{X}||^2 - ||\mathcal{G}||^2 \leq \epsilon^2 \cdot ||\mathcal{X}||^2$$

$$\therefore ||\mathcal{X}||^2 - \epsilon^2 \cdot ||\mathcal{X}||^2 \leq ||\mathcal{G}||^2$$

$$\therefore (1 - \epsilon^2) \cdot ||\mathcal{X}||^2 \leq ||\mathcal{G}||$$

We can thus estimate the relative error in the approximation by computing $||\mathcal{G}||^2$ and choosing the next rank-$\mathbf{R}$ so that $||\mathcal{G}(1:\mathbf{R})||^2 \approx (1-\epsilon^2)||\mathcal{X}||^2$. Specifically, we solve the optimization problem

$$\underbrace{\arg\min}_{\mathbf{R}} ||\mathcal{G}(1:\mathbf{R})||^2$$

$$\text{subject to } ||\mathcal{G}(1:\mathbf{R})||^2 \geq (1-\epsilon^2)||\mathcal{X}||^2$$

The details of this algorithm, which we call Adaptive HOOI, can be seen in Algorithm 3. 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.

The work done in the first project is more extensive. Recall that we can actively search for fast matrix multiplication algorithms by optimization as in Figure 1.6. We can do so by performing an CP Decomposition on the matrix multiplication tensor by specifying the size of the matrix n and the number of multiplications in the algorithm R. This is described in algorithm 5 where the details of how to compute the Function value, the gradient and the Jacobian are left as an exercise to the reader. But now, how do we modify it so that we can decrease the number of possibilities from 3^{84} to

3^{28} by relying on the cyclic invariance structure described earlier? The details of so can be seen in Algorithm 6. I once again leave the now much, much more complicated details of how to compute the function value, the gradient and the Jacobian as another exercise to the reader.

3.3 Expected Outcomes

The goal for the first project is difficult to describe. Preliminary work on attempting to search for fast matrix multiplication algorithm shows that it is a daunting task. Even though exhaustive search is not enough to prove that an algorithm doesn't exist for certain ranks, it is clear that there are many more places where an algorithm fails to exist than the ones where it actually exists. But have implemented a new way of searching these algorithms much more efficiently than what was done in the past, and it could prove itself worthy of future use. We have more recently turned to a more mathematical approach to the project with the aid of Dr. Frank Moore and Dr. Pratyush Mishra. But where we are heading is still unknown. We wish to share some both the new method of searching for algorithm and some of the interesting solutions we found.

The goals of the second and third projects are simpler. We plan to have a fully working implementation of Adaptive HOOI and TTSVD in MATLAB and the TuckerMPI. Additionally, we wish to demonstrate that Adaptive HOOI is more efficient

than STHOSVD given the certain configurations, and that TTSVD is more efficient than both when the number of the dimensions of the input tensor is large. We wish to test large datasets such as the Miranda, HCCI, and SP Tensor.

3.4 Algorithms

Algorithm 1 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**

$[U_k, \epsilon_k] \leftarrow \text{LLSV}(G_{(k)}, r_k \text{ or } \bar{\epsilon})$ ▷ 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

Algorithm 2 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

Algorithm 3 Adaptive 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
Rank Size Increase Rate α
Output: TTensor $\mathcal{T}$ of ranks $\mathbf{R}$ with $\mathcal{T} \approx \mathcal{X}$

function AdaptiveHOOI($\mathcal{X}$, $\mathbf{R}$ 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

$\mathcal{G} \leftarrow \mathcal{Y} \times_d U_d^\intercal$ ▷ update core

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 $||\mathcal{G}(1 : \mathbf{R})||^2 \geq (1 - \epsilon^2)||\mathcal{X}||^2$

$\mathcal{G} = \mathcal{G}(1 : \mathbf{R}, U_k = U_k(1 : r_k)\ \forall k \in [d]$

else

$\mathbf{R} = \alpha\ \mathbf{R}$

end if

end for

return $[\mathcal{G}, U_{1:d}]$ ▷ $\mathcal{T} \equiv \{\mathcal{G}; U_{1:d}\}$

end function

Algorithm 4 TTSVD

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** $\bar{\epsilon} \leftarrow (\epsilon/\sqrt{d-1}) \cdot ||\mathcal{X}||$

$r_0 \leftarrow 1$

for $k = 1, \ldots, d-1$ **do**

if $k == 1$ **then**

$\bar{\mathbf{Y}}_1 \leftarrow \mathbf{X}_{(1)}$

else

$\bar{\mathbf{Y}}_k \leftarrow \text{reshape}(\mathbf{Y}_k, (r_{k-1} n_k) \times (n_{k+1} \cdots nd)$

end if

$[\mathbf{G}_k, \epsilon_k] \leftarrow \text{LLSV}(\bar{\text{Y}}_k, r_k \text{ or } \bar{\epsilon})$

$\mathbf{Y}_{k+1} \leftarrow \mathbf{G}_k^\intercal \bar{\mathbf{Y}}_k$

$\mathcal{G}_k \leftarrow \text{reshape}(\mathbf{G}_k, r_{k-1} \times n_k \times r_k)$

end for

$\mathcal{G}_d \leftarrow \mathbf{Y}_d$

$\text{ERR} \leftarrow \sqrt{\sum_{k=1}^{d-1} \epsilon_k^2}$ ▷ equivalent to $||\mathcal{X} - \mathcal{T}||$

return $[\mathcal{G}_{1:d}, \text{ERR}]$ ▷ $\mathcal{X} \equiv \{\mathcal{G}_\infty, \cdots, \mathcal{G}_\lceil\}$

end function

Algorithm 5 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_old $\longleftarrow \frac{1}{2}\|\mathcal{M} - [A, B, C]\|$ ▷ Compute Function Value

$\nabla F \longleftarrow [\text{vec}\left(\frac{\partial f}{\partial A}\right) \text{vec}\left(\frac{\partial f}{\partial B}\right) \text{vec}\left(\frac{\partial f}{\partial C}\right)]^T$ ▷ Compute Gradient of Function

$S \longleftarrow$ Solution to$(\mathbf{J}^T\mathbf{J} + \lambda I)K$ ▷ Where $\mathbf{J}$ is the Jacobian

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_old - F_new $< \epsilon$ **then**

break

end if

end for

end function

Algorithm 6 Damped Gauss-Newton On The Matrix Multiplication Tensor But Better

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 $n^2 \times Rs$ and the remaining three $n^2 \times Rc$ matrices

for $i = 1 : MaxIters$ **do**

F_old $\longleftarrow \frac{1}{2}\|\mathcal{M} - [S, S, S] - [U, V, W] - [W, U, V] - [V, W, U]\|$

$\nabla F \longleftarrow [\text{vec}\left(\frac{\partial f}{\partial S}\right) \text{vec}\left(\frac{\partial f}{\partial U}\right) \text{vec}\left(\frac{\partial f}{\partial V}\right) \text{vec}\left(\frac{\partial f}{\partial W}\right)]^T$

$S \longleftarrow$ Solution to$(\mathbf{J}^T\mathbf{J} + \lambda I)K$ ▷ Where **J** is the Jacobian

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_old - F_new $< \epsilon$ **then**

break

end if

end for

end function

Bibliography

[1] Thomas Lam. *100 Questions: A Mathematical Conventions Survey*. NYU Courant Institute of Mathematical Sciences. 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] Rachel Minster, Zitong Li, and Grey Ballard. "Parallel Randomized and Tucker Decomposition Algorithms". In: *SIAM Journal on Scientific Computing* 46.2 (2024), A1186–A1213.

[4] Rachel Minster, Zitong Li, and Grey Ballard. "Parallel Randomized Tucker Decomposition Algorithms". In: *arXiv* 2211.13028 (2022).

[5] Zitong Li, Qiming Fang, and Grey Ballard. "Parallel Tucker Decomposition with Numerically Accurate SVD". In: *ICPP '21: Proceedings of the 50th International Conference on Parallel Processing* 49 (2021), pp. 1–11.

[6] Tamara G. Kolda, Alicia Klinvex, and Grey Ballard. "TuckerMPI: A Parallel C++/MPI Software Package for Large-Scale Data Compression via the Tucker Decomposition". In: *ACM Transactions on Mathematical Software (TOMS)* 46.13 (2 2020), pp. 1–31.

[7] Grey Ballard. "CP and Tucker Tensor Decompositions". In: *SIAG/OPT Views and News* 27.1 (2019), pp. 1–7.

[8] Tamara G. Kolda, Alicia Klinvex, and Grey Ballard. "uckerMPI: Efficient parallel software for Tucker decompositions of dense tensors". In: *arXiv* 1901.06043 (2019).

[9] Hussam Al Daas et al. "Tight Memory-Independent Parallel Matrix Multiplication Communication Lower Bounds". In: *ACM* 2 (2022).

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

[11] Kangning Cui et al. *PalmProbNet: A Probabilistic Approach to Understanding Palm Distributions in Ecuadorian Tropical Forest via Transfer Learning*. 2024. arXiv: 2403.03161 [cs.CV]. URL: https://arxiv.org/abs/2403.03161.