

High-Performance Sparse Tensor Factorization: Algorithms for Hyper-Sparse
Streaming Data and Scalable Non-Negative Least Squares Solvers

by

Yongseok Paul Soh

A dissertation accepted and approved in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science

Dissertation Committee:

Jee Choi, Chair

Hank Childs, Core Member

Thanh Nguyen, Core Member

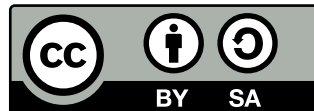
Jake Searcy, Institutional Representative

University of Oregon

Fall 2025

© 2025 Yongseok Paul Soh

This work, including text and images of this document but not including supplementary files (for example, not including software code and data), is openly licensed via CC BY-SA 4.0.



DISSERTATION ABSTRACT

Yongseok Paul Soh

Doctor of Philosophy in Computer Science

Title: High-Performance Sparse Tensor Factorization: Algorithms for Hyper-Sparse Streaming Data and Scalable Non-Negative Least Squares Solvers

Tensor factorization is a fundamental tool for analyzing multi-dimensional data, a process critical for applications ranging from recommender systems to scientific discovery. However, its practical application is often limited by severe performance bottlenecks. In this talk, we address the specific computational challenges that arise when factorizing real-world sparse tensors, which are characterized by irregular memory access and significant memory overhead. To overcome these issues, we present novel optimizations that include reformulating core computations and developing dynamic data encoding strategies, such as tensor linearization and adaptive time slicing, to efficiently process time-evolving sparse data. Furthermore, we address the additional challenges that arise in the constrained variant of sparse tensor factorization, where enforcing properties like non-negativity can introduce significant computational overhead. To solve this, we present our specialized optimizations that leverage hardware-specific optimizations, demonstrating how to efficiently solve constrained factorization problems at scale.

This dissertation includes previously published and unpublished manuscripts and coauthored material.

CURRICULUM VITAE

NAME OF AUTHOR: Yongseok Paul Soh

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA

Seoul National University, Seoul, Republic of Korea

Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea

DEGREES AWARDED:

Doctor of Philosophy, Computer Science, 2025, University of Oregon

Master of Science, Electrical and Computer Engineering, 2012, Seoul National University

Bachelor of Science, Electrical Engineering, 2010, Korea Advanced Institute of Science and Technology

AREAS OF SPECIAL INTEREST:

High Performance Computing

Sparse Tensor Factorization

Constrained Optimization

PROFESSIONAL EXPERIENCE:

Graduate Research/Teaching Assistant, University of Oregon, 2019-2025

Graduate Research Intern, Oak Ridge National Laboratory, Discrete Algorithms, January 2025 - March 2025

Graduate Research Intern, Oak Ridge National Laboratory, Discrete Algorithms, July 2023 - September 2023

Graduate Research Intern, Intel Corporation, Parallel Computing Lab, June 2020 - September 2020

Machine Learning Engineer, Linewalks, Inc, Research and Development Team, March 2017 - August 2019

Software Engineer, Tumblbug, Inc, Product Team, May 2016 - March 2017

Systems Engineer, Hyundai Motor Company, Commercial Vehicle Design Center, January 2012 - May 2015

GRANTS, AWARDS, AND HONORS:

J. Donald Hubbard Family Scholarship, 2025

Gurdeep Pall Graduate Student Fellowship, 2021

Erwin & Gertrude Juilfs Scholarship, 2020

Lokey Graduate Science Award, 2019

Student Volunteer, Supercomputing Conference, 2020, 2022, 2024

Travel Grant, Supercomputing Conference, 2022, 2024

PUBLICATIONS:

Yongseok Soh, Ahmed E. Helal, Fabio Checconi, Jan Laukemann, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jee W. Choi, *Dynamic Tensor Linearization and Time Slicing for Efficient Factorization of Infinite Data Streams*, 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, USA, 2023, 402–412, DOI: 10.1109/IPDPS54959.2023.00048, DOI: 10.1109/IPDPS54959.2023.00048

Yongseok Soh, Patrick Flick, Xing Liu, Shaden Smith, Fabio Checconi, Fabrizio Petrini, and Jee Choi, *High Performance Streaming Tensor Decomposition* (en), 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, Portland, OR, USA, 2021, 683–692, DOI: 10.1109/IPDPS49936.2021.00078

Yongseok Soh, Ramakrishnan Kannan, Piyush Sao, and Jee Choi, *Accelerated Constrained Sparse Tensor Factorization on Massively Parallel Architectures*, Proceedings of the 53rd International Conference on Parallel Processing, Association for Computing Machinery, New York, NY, USA, 2024, 107–116

Andy Nguyen, Ahmed E. Helal, Fabio Checconi, Jan Laukemann, Jesmin Jahan Tithi, Yongseok Soh, Teresa Ranadive, Fabrizio Petrini, and Jee W. Choi, *Efficient, Out-of-Memory Sparse MTTKRP on Massively Parallel Architectures* (en), Proceedings of the 36th ACM International Conference on Supercomputing, ICS '22, Association for Computing Machinery, Virtual Event, 2022, 1–13, DOI: 10.1145/3524059.3532363, arXiv:2201.12523 [cs]

Jan Laukemann, Ahmed E. Helal, S. Isaac Geronimo Anderson, Fabio Checconi, Yongseok Soh, Jesmin Jahan Tithi, Teresa Ranadive, Brian J. Gravelle, Fabrizio Petrini, and Jee Choi, *Accelerating Sparse Tensor Decomposition Using Adaptive Linearized Representation*, IEEE Transactions on Parallel and Distributed Systems **36** (2025), no. 5, 1025–1041, DOI: 10.1109/TPDS.2025.3553092, DOI: 10.1109/TPDS.2025.3553092

TABLE OF CONTENTS

Chapter	Page
LIST OF FIGURES	12
LIST OF TABLES	15
1. INTRODUCTION	16
1.1. Research Questions	18
1.2. Dissertation Statement	18
2. BACKGROUND	20
2.1. Sparse Tensors	20
2.1.1. <i>Hyper-Sparse Tensors</i>	20
2.1.2. <i>Sparse Tensor Datasets</i>	21
2.2. Tensor Operations	21
2.2.1. <i>Tensor Matricization</i>	22
2.2.2. <i>Khatri-Rao Product</i>	22
2.2.3. <i>Matricized Tensor Times Khatri-Rao Product (MTTKRP)</i> . .	23
2.2.4. <i>Distinction between Dense and Sparse MTTKRP</i>	23
2.3. Sparse Tensor Representations	25
2.3.1. <i>List-based Formats</i>	26
2.3.2. <i>Tree-based Formats</i>	27
2.3.3. <i>Block-based Formats</i>	30
2.3.4. <i>Linearized Formats</i>	31
2.4. Tensor Factorization	34
2.4.1. <i>Canonical polyadic (CP) decomposition</i>	35

3. ROW-SPARSE CP-STREAM	38
3.1. Problem Statement	38
3.2. Tensor Factorization for Streaming Sparse Tensors	38
3.2.1. <i>Streaming Sparse Tensors</i>	39
3.2.2. <i>Challenges in Streaming Sparse Tensor Factorization</i>	39
3.2.3. <i>CP-stream</i>	41
3.3. Methodology: Row-Sparse CP-stream	42
3.3.1. <i>Motivation</i>	42
3.3.2. <i>spCP-stream Notations</i>	44
3.3.3. <i>spCP-stream Algorithm</i>	44
3.3.4. <i>Pre- and Post- Operations</i>	47
3.3.5. <i>Convergence Check</i>	47
3.4. Performance Evaluation and Analysis	48
3.4.1. <i>Datasets and Execution Parameters</i>	49
3.4.2. <i>Execution Time, Fit Error, and Convergence Property</i>	49
3.4.3. <i>Performance Analysis of Unconstrained CP-stream</i>	50
3.5. Conclusion	54
4. STREAMING CPD WITH ADAPTIVE LINEARIZED TENSOR ORDER- ING	57
4.1. Problem Statement	57
4.2. Motivation	57
4.3. Methodology: ALTO-stream	59
4.3.1. <i>Spatial Sketching</i>	61
4.3.2. <i>Dynamic Index Linearization</i>	64
4.3.3. <i>Dynamic Time Slicing</i>	68

4.3.4.	<i>Putting It All Together: ALTO-stream</i>	70
4.4.	Evaluation	70
4.4.1.	<i>Experimental Setup</i>	72
4.4.2.	<i>Results</i>	74
4.4.3.	<i>Model Validation</i>	78
4.5.	Conclusion	79
5.	FULLY GPU-ACCELERATED CONSTRAINED SPARSE TENSOR FAC-	
	TORIZATION	81
5.1.	Problem Statement	81
5.2.	Motivation	81
5.3.	Methodology: Fully GPU-Accelerated Constrained Sparse Tensor Fac-	
	torization	84
5.3.1.	<i>Profiling the baseline cSTF implementation</i>	85
5.3.2.	<i>cuADMM - GPU-optimized ADMM algorithm</i>	86
5.3.3.	<i>Optimized ADMM updates Implementation</i>	87
5.4.	Evaluation	90
5.4.1.	<i>Optimized ADMM performance analysis</i>	90
5.4.2.	<i>Comparison against the state-of-the-art on the latest NVIDIA GPUs</i>	91
5.4.3.	<i>Accelerating additional non-negativity constraint algorithms</i>	95
5.5.	Conclusion	96
6.	SCALABLE NONNEGATIVE LEAST SQUARES SOLVERS	99
6.1.	Problem Statement	99
6.2.	Accelerated ADMM on Multicore Shared Memory Systems	101
6.2.1.	<i>NNLS Formulation</i>	101
6.2.2.	<i>Computational and Memory Access Cost for ADMM</i>	102

6.2.3.	<i>Blocked and Fused ADMM</i>	104
6.2.4.	<i>Performance Results for Optimized ADMM on Multicore Shared Memory Systems</i>	105
6.3.	Active-set Based Methods	106
6.3.1.	<i>NNLS Formulation</i>	108
6.3.2.	<i>The Block Principal Pivoting (BPP) Approach</i>	109
6.3.3.	<i>Performance Analysis of the Mask Constrained Normal Equation Solver</i>	111
6.3.4.	<i>Problem Formulation</i>	112
6.3.5.	<i>Limitations of Direct Factorization-based Solvers for Multiple RHS</i>	114
6.3.6.	<i>Limitations of Traditional Iterative Solvers for Constrained Prob- lems</i>	115
6.4.	FastBPP: Proposed Framework	116
6.4.1.	<i>Block Masked Preconditioned CG</i>	116
6.4.2.	<i>Convergence Analysis</i>	117
6.4.3.	<i>Efficacy of Global Preconditioning for PBCG</i>	119
6.4.4.	<i>Sparsity-based Adaptive Block Masked CG</i>	120
6.4.5.	<i>Maintaining Optimality with Direct Solver Verification</i>	121
6.4.6.	<i>Performance Results for Block Principal Pivoting on Massively Parallel GPU Systems</i>	123
6.5.	Conclusion	128
7.	CONCLUSION AND FUTURE WORK	129
7.1.	Conclusion	129
7.2.	Key Contributions by Research Questions	129
7.3.	Future Research Directions	134

7.3.1. <i>Scaling up Constrained Factorization Algorithms for Domain Science Applications</i>	134
7.3.2. <i>Data-Driven Space Filling Curves for Sparse Tensors</i>	136
BIBLIOGRAPHY	139

LIST OF FIGURES

Figure		Page
2.1.	Matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ to a matrix $\mathbf{X}_{(1)}$	22
2.2.	Rank- R CPD of a third-order tensor	36
3.1.	Illustration of streaming sparse tensors	41
3.2.	Histogram of non-zero elements for the <i>Flickr</i> dataset at time slice 500 .	43
3.3.	Comparison of <i>spCP-stream</i> and optimized non-constrained CP-stream to the baseline on the <i>NIPS</i> dataset.	51
3.4.	Comparison of <i>spCP-stream</i> and optimized non-constrained CP-stream to the baseline on the <i>Patents</i> , <i>Uber</i> , and <i>Flickr</i> datasets	52
3.5.	Execution time breakdown for the <i>Flickr</i> dataset	53
4.1.	Density trend for the Chicago and LANL streaming tensors	58
4.2.	Overview of our ALTO-stream framework.	60
4.3.	Illustration of applying spatial sketching, dynamic index linearization, and dynamic time slicing to a set of time slices from the Chicago tensor . . .	63
4.4.	Illustration of output-oriented traversal of nonzero elements	66
4.5.	The performance of CP-stream algorithm using ALTO-stream compared to the state-of-the-art SPLATT-stream	72
4.6.	The performance of spCP-stream algorithm using ALTO-stream compared to the state-of-the-art SPLATT-stream	73
4.7.	Varying number of nonzero elements across temporal tensor slices	75
4.8.	The performance of spCP-stream algorithm using ALTO-stream for varying factorization ranks	77

4.9. Comparison of the global relative fitting error between SPLATT-stream and ALTO-stream, with and without DTS	79
5.1. Execution time breakdown of constrained tensor factorization for dense and sparse tensors	82
5.2. Execution time breakdown of cSTF on three tensors with the largest number of non-zero elements	85
5.3. Speedup achieved by cuADMM over the baseline ADMM implementation	90
5.4. End-to-end per iteration speedup vs. SPLATT using ADMM update for rank $R = 32$ on A100 GPU	92
5.5. End-to-end per iteration speedup vs. SPLATT using ADMM update for rank $R = 32$ on H100 GPU	93
5.6. Comparison of speedup achieved for MTTKRP and ADMM across 10 real-world sparse tensors on the A100 GPU	94
5.7. Comparison of speedup achieved for MTTKRP and ADMM across 10 real-world sparse tensors on the H100 GPU	94
5.8. Speedup achieved on the A100 GPU by our GPU cSTF framework using the MU and HALS update schemes against PLANC, the state-of-the-art CPU-based cSTF library.	96
5.9. Speedup achieved on the H100 GPU by our GPU cSTF framework using the MU and HALS update schemes against PLANC, the state-of-the-art CPU-based cSTF library.	97
6.1. Motivational plot brought from [23]	101
6.2. Speedup achieved by our Blocked & Fused <i>ADMM</i> and Hybrid Lock <i>MTTKRP</i> over the baseline	107

6.3. Comparison of Blocked & Fused <i>ADMM</i> kernel to the baseline on the NIPS dataset.	107
6.4. Execution time breakdown of Active-set NNLS Solver (Block Pivoting Principle (BPP) Method) for problems with varying scale	111
6.5. Illustration of the mask constrained normal equation problem	112
6.6. Direct vs. iterative solve times for mask constrained normal equations under various scaling conditions	114
6.7. Preconditioning effectiveness for two distinct preconditioning strategies . .	119
6.8. Performance comparison of Block Masked CG based solvers for high column sparsity (>0.5)	120
6.9. Speedup of FastBPP and FastBPP-V over a baseline method	123
7.1. Spatiotemporal evolution of structural phase transitions in a 2D material lattice	135

LIST OF TABLES

Table	Page
2.1. Real-world sparse tensor data sets.	21
2.2. Comparison of sparse tensor formats	25
2.3. Overview comparison of CP-Decomposition, Tucker Decomposition, and Tensor- Train Decomposition	34
3.1. The target streaming sparse tensors, ordered by the number of nonzero el- ements	39
3.2. Datasets used in spCP-stream evaluation	49
4.1. The target streaming sparse tensors for ALTO-stream evaluation	72
4.2. Memory metrics (loads/stores) of spCP-stream across the STF frameworks	78
5.1. Hardware and software setup for the evaluation in Chapter 5	89
6.1. Compute and memory costs for different ADMM operations.	104
6.2. A comparison of FLOPS for different subproblem solvers	113
6.3. Performance comparison of FastBPP variants	125
6.4. Solver accuracy and BPP iteration count comparison of FastBPP variants	126

CHAPTER 1

INTRODUCTION

Modern applications routinely generate and collect large and complex multi-modal data that naturally exhibit higher-order relationships beyond simple pairwise interactions. From social network platforms tracking user-item-context-time interactions, healthcare systems monitoring patient-treatment-outcome-temporal patterns, or cybersecurity platforms analyzing network flows, these information-rich datasets capture intricate, multi-faceted dependencies. Tensors which are the mathematical generalization of matrices to higher orders, provide a natural and powerful abstraction for representing and analyzing such multi-dimensional data.

Unraveling the inherent intricate underlying patterns in these multi-dimensional tensors are an essential yet challenging problem, where *tensor factorization* algorithms are a key tool to uncover the latent factors. However, scaling such factorization algorithms to handle large scale tensors across different computational domains presents significant challenges that needs to be thoroughly addressed.

The challenges can be summarized as follows:

First, real-world tensors are often extremely *sparse*, with the vast majority of possible interactions never occurring in practice. For instance, in recommendation systems, only a small fraction of users interact with a small subset of available items; in network traffic analysis, most source-destination-protocol combinations remain unobserved. For factorization algorithms to handle these extreme sparsity demands specialized algorithms and data structures that can efficiently process billions of nonzero elements scattered across exponentially larger tensor spaces.

Second, many critical applications generate data in a *streaming* manner, where new interactions continuously arrive over a time. Financial transactions, social me-

dia updates, sensor readings, and network logs all exemplify infinite data streams that can potentially evolve temporally. Traditional tensor factorization methods, designed for static datasets, fail to accommodate this unbounded, time-varying nature of streaming data. The challenge intensifies as the tensor space expands with each new time slice, requiring algorithms that can incrementally update factorizations without repeatedly processing all historical data.

Third, the sheer *scale* of modern real world tensor data can easily overwhelm conventional computing systems. Tensors with billions or even trillions of nonzero elements are increasingly common, yet they must be processed within stringent time and memory constraints. Extracting latent features through tensor factorization demands high-performance computing techniques that can effectively harness modern parallel architectures, including multi-core CPUs and massively parallel GPUs. The memory bandwidth limitations, synchronization overhead, and load imbalance inherent in sparse tensor algorithms pose fundamental obstacles to achieve efficient computation.

Fourth, many applications require *constrained* tensor factorizations that incorporate domain-specific knowledge to produce interpretable and physically meaningful results. Specifically, non-negativity constraints are essential when analyzing data such as count data, signal strength, or probabilities; However, enforcing such constraints significantly increases computational complexity, as each iteration must solve a constrained optimization problem rather than a simple least-squares update.

The combination of these challenges—extremely sparse and dynamic streaming tensors at massive scale, and constrained optimization required to handle such tensors— motivates the need for innovative algorithmic and systems-level solutions.

This dissertation addresses this critical gap by developing high-performance methods for streaming sparse and constrained sparse tensor factorization that are

specifically designed for modern parallel computing architectures and streaming data environments.

1.1 Research Questions

In this dissertation, we address the following research questions:

RQ1: How can we optimize streaming sparse tensor factorization throughput by exploiting the hyper-sparse nature of the streaming tensors?

RQ2: Can we further improve the throughput of streaming sparse tensor factorization utilizing adaptive linearized sparse tensor formats?

RQ3: With the advent of powerful GPUs, can we utilize these parallel computing architectures by implementing a fully GPU-resident constrained sparse tensor factorization framework?

RQ4: Can we accelerate the constrained optimization problem in constrained sparse tensor factorization by exploiting the inherent computational structure?

1.2 Dissertation Statement

This dissertation advances the state-of-the-art in high-performance constrained sparse tensor factorization through novel algorithmic innovations, adaptive data structures, and hardware-aware optimizations for modern parallel computing architectures. In Chapter 2, we lay out the preliminary background on sparse tensors and tensor factorization. In Chapter 3 and 4, we present a novel algorithm and adaptive data structure for streaming sparse tensor factorization that exploits the

hyper-sparse nature of streaming tensors. In Chapter 5, we present a fully GPU-resident sparse tensor factorization framework that has support for constrained factorization. In Chapter 6, we present scalable and efficient NNLS solvers that are crucial for supporting constrained factorization at scale.

Co-authored and Published Papers Disclaimer Parts of this dissertation (Chapters 3, 4, 5, and 6) are adapted from published and unpublished papers and are a result of collaboration with co-authors: Patrick Flick, Xing Liu, Shaden Smith, Fabio Checconi, Fabrizio Petrini, Ahmed E. Helal, Jan Laukemann, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, Ramakrishnan Kannan, Piyush Sao and Professor Jee Choi. More detail on the breakdown of the work for those publications will be stated at the start of each chapter.

CHAPTER 2

BACKGROUND

In this chapter, we provide a comprehensive overview of the background of the dissertation. We will lay out the preliminary knowledge of sparse tensors and then move on to tensor operations, sparse tensor representations, and sparse tensor factorization algorithms.

2.1 Sparse Tensors

Tensors are higher-order generalization of matrices, and they provide a natural abstraction for complex and inter-related data. Many critical applications, such as data mining [47, 78], social network analytics [80, 25], cybersecurity [24, 11], and healthcare [33, 31], generate massive amounts of multi-dimensional (multi-modal) data. These tensors are inherently sparse by nature because the number of observed interactions is vastly outnumbered by the exponential space of all possible combinations across multiple dimensions. Such datasets can naturally be represented as sparse tensors, which store the values of the nonzero tensor elements along with indexing metadata that denote the position of each nonzero in the tensor.

2.1.1 Hyper-Sparse Tensors

Typically, sparse tensors are said to be *hyper-sparse* if the number of nonzero elements is significantly less than the total number of elements in the tensor. However, for the scope of this dissertation, we adopt a more specific definition and use the term *hyper-sparse* to refer to tensors where the non-zero elements do not span across the entire tensor space. This specialized definition is safe to presume, as ten-

Table 2.1. Real-world sparse tensor data sets.

Tensor	Dimensions	#NNZs	Density
LBNL	$1.6K \times 4.2K \times 1.6K \times 4.2K \times 868.1K$	$1.7M$	4.2×10^{-14}
NIPS	$2.5K \times 2.9K \times 14K \times 17$	$3.1M$	1.8×10^{-06}
UBER	$183 \times 24 \times 1.1K \times 1.7K$	$3.3M$	3.8×10^{-04}
CHICAGO	$6.2K \times 24 \times 77 \times 32$	$5.3M$	1.5×10^{-02}
VAST	$165.4K \times 11.4K \times 2 \times 100 \times 89$	$26M$	7.8×10^{-07}
DARPA	$22.5K \times 22.5K \times 23.8M$	$28.4M$	2.4×10^{-09}
ENRON	$6K \times 5.7K \times 244.3K \times 1.2K$	$54.2M$	5.5×10^{-09}
LANL-2	$3.8K \times 11.2K \times 8.7K \times 75.2K \times 9$	$69.1M$	1.9×10^{-10}
NELL-2	$12.1K \times 9.2K \times 28.8K$	$76.9M$	2.4×10^{-05}
FB-M	$23.3M \times 23.3M \times 166$	$99.6M$	1.1×10^{-09}
FLICKR	$319.7K \times 28.2M \times 1.6M \times 731$	$112.9M$	1.1×10^{-14}
DELI	$532.9K \times 17.3M \times 2.5M \times 1.4K$	$140.1M$	4.3×10^{-15}
NELL-1	$2.9M \times 2.1M \times 25.5M$	$143.6M$	9.1×10^{-13}
AMAZON	$4.8M \times 1.8M \times 1.8M$	$1.7B$	1.1×10^{-10}
PATENTS	$46 \times 239.2K \times 239.2K$	$3.6B$	1.4×10^{-03}
REDDIT	$8.2M \times 177K \times 8.1M$	$4.7B$	4.0×10^{-10}

sors exhibiting such localized sparsity patterns are inherently extremely sparse, satisfying the traditional definition as well. This is typically encountered in *streaming* tensors, where the non-zero elements for a given time slice are likely to not be populated across all the modes.

2.1.2 Sparse Tensor Datasets

As many real-world applications generate sparse tensors, there has been several benchmark datasets for sparse tensors. For the scope of this dissertation, we will focus on the following datasets provided by [85].

2.2 Tensor Operations

There are a wide variety of tensor operations that are used in various tensor algorithms. For the scope of this dissertation, we will focus on the tensor opera-

tions that are relevant to a specific tensor factorization algorithm: the Canonical Polyadic Decomposition (CPD) [14, 30, 68].

2.2.1 Tensor Matricization

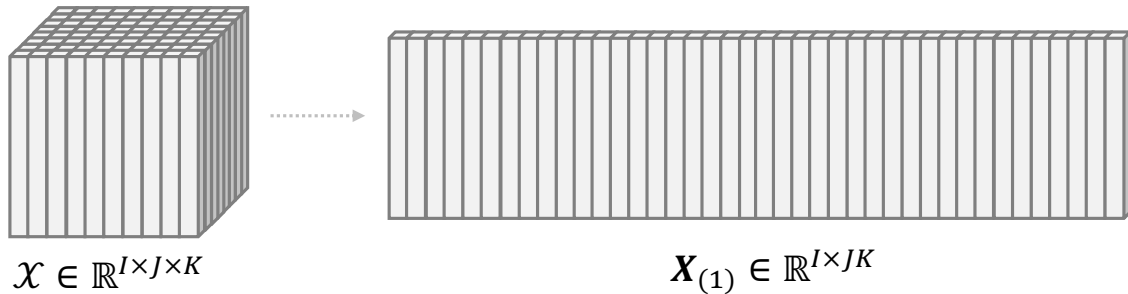


Figure 2.1. Matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ to a matrix $\mathbf{X}_{(1)}$.

Matricization is the process of flattening a tensor into a matrix and is a fundamental operation in tensor factorization. In tensors, we use the term *fiber* to refer to the set of elements that are indexed by a fixed value of a given mode. In matrix terminology, a tensor fiber can be thought of as a column of the matrix. Figure 2.1 illustrates the matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ to a matrix $\mathbf{X}_{(1)}$.

2.2.2 Khatri-Rao Product

The Khatri-Rao product is a generalization of the Kronecker product to matrices with matching column dimensions [82, 59]. It is defined as the column-wise Kronecker product of two matrices. For two matrices $\mathbf{A} \in \mathbb{R}^{I \times K}$ and $\mathbf{B} \in \mathbb{R}^{J \times K}$, the Khatri-Rao product is given by:

$$\mathbf{A} \odot \mathbf{B} = (\mathbf{a}_1 \otimes \mathbf{b}_1, \mathbf{a}_2 \otimes \mathbf{b}_2, \dots, \mathbf{a}_K \otimes \mathbf{b}_K) \quad (2.1)$$

where the resulting product is of size $\mathbb{R}^{IJ \times K}$.

2.2.3 Matricized Tensor Times Khatri-Rao Product (MTTKRP)

A critical and computationally intensive component of CP-ALS, as well as numerous other tensor algorithms, is the matricized tensor times Khatri-Rao product (MTTKRP) operation [3]. The MTTKRP operation combines the two operations previously described:

1. *Tensor matricization* is the process by which a tensor is *unfolded* into a matrix. The mode- n matricization of a tensor $\mathcal{X}$, denoted $\mathbf{X}_{(n)}$, is obtained by laying out the mode- n fibers of $\mathcal{X}$ as the columns of $\mathbf{X}_{(n)}$.
2. *Khatri-Rao product* is the “matching column-wise” Kronecker product between two matrices. That is, given matrices $\mathbf{B} \in \mathbb{R}^{J \times R}$ and $\mathbf{C} \in \mathbb{R}^{K \times R}$, their Khatri-Rao product $\mathbf{K}$, denoted $\mathbf{K} = \mathbf{B} \odot \mathbf{C}$, where $\mathbf{K}$ is a $(J \cdot K) \times R$ matrix, is defined as: $\mathbf{B} \odot \mathbf{C} = [\mathbf{b}_1 \otimes \mathbf{c}_1 \ \mathbf{b}_2 \otimes \mathbf{c}_2 \ \dots \ \mathbf{b}_R \otimes \mathbf{c}_R]$.

2.2.4 Distinction between Dense and Sparse MTTKRP

The MTTKRP operation has drastically different performance implications depending on the density (or sparsity) of the data tensor of interest. Expanding on the notion of dense and sparse matrices, dense tensors are entities where all entries are filled with numerical values, which often encapsulates fully populated multidimensional arrays. For instance, a three-dimensional tensor representing RGB values for every pixel in a set of images can be considered a dense tensor as it would have all of its elements filled.

In contrast, sparse tensors are entities that contain a significantly smaller proportion of non-zero values relative to their total number of elements. These structures are particularly common in domains where presence rather than absence is

of interest, such as social networks represented in a tensor with dimensions for users, friends, and time, where an entry is only non-zero if a connection was made at a specific time. Another example is in scientific computing, where sparse tensors might represent three-dimensional simulation data with most values being zero except where specific phenomena, such as high-stress regions in a material, are present.

In the domain of dense MTTKRP computation, the operations are carried out explicitly. The process entails formulating the matricized tensor and the Khatri-Rao product within the memory and employing a matrix-multiply kernel to execute the MTTKRP product. For multi-dimensional tensors, recent studies have explored workload partitioning strategies to enable the reuse of partial results [23, 34, 39], thereby enhancing computational efficiency. Nonetheless, the memory overhead remains substantial and represents the predominant cost associated with constrained dense tensor factorization.

Conversely, sparse MTTKRP computations circumvent the need for explicitly forming all intermediate products. This method involves iterating through the non-zero entries of a tensor to collect the required data for calculating the MTTKRP result for each non-zero element. Theoretically, this approach lends itself to acceleration via the parallelization of workloads aligned with the partition of non-zero entries. Yet, the implementation of such parallelization must contend with challenges including high synchronization costs and the impact of memory access patterns on efficiency. Significant research efforts, such as those outlined in [32, 67], have been dedicated to devising efficient schemes for sparse tensor representations. These initiatives focus on multiple objectives, including the optimization of workload partitioning, enhancement of data locality, among other considerations critical to the advancement of sparse tensor factorization techniques. In this dissertation, we will

Format	Granularity	Mode Spec	Idx Comp	Data Local	Upd Conflicts	Work Bal
List based						
COO [17]	NZ element	Agnostic	×	×	×	✓
F-COO [63]	NZ element	Specific	×	×	✓	✓
Tree-based						
CSF [90]	Subtree	Specific	✓	✓	✓	×
B-CSF [70]	Hybrid	Specific	✓	✓	✓	×
MM-CSF [69]	Subtree	Agnostic	✓	✓	✓	×
Block-based						
HiCOO [52]	Subblock	Agnostic	✓	✓	×	×
Reordered HiCOO [53]	Subblock	Agnostic	✓	✓	×	×
Linearized						
LCO [29]	NZ element	Agnostic	✓	✓	✓	✓
ALTO [32]	NZ element	Agnostic	✓	✓	✓	✓
BLCO [67]	NZ element	Agnostic	✓	✓	✓	✓

Table 2.2. Comparison of sparse tensor formats. *Granularity*: granularity of data organization. *Mode Spec*: mode-specific or mode-agnostic. *Idx Comp*: index compression capability. *Data Local*: data locality optimization. *Upd Conflicts*: mechanisms to reduce update conflicts. *Work Bal*: workload balance characteristics. See Section 2.3 for detailed discussions.

focus exclusively on sparse tensors and their related algorithms for the remainder of the discussion.

2.3 Sparse Tensor Representations

The sparsity of tensors presents significant challenges for efficient storage and computation. As the size and dimensionality of datasets continue to grow, the need for optimized sparse tensor representations becomes increasingly critical. Traditional dense tensor formats are impractical for sparse tensors due to excessive memory requirements, prompting researchers to develop specialized formats and algorithms for sparse tensor operations. However, designing an optimal sparse tensor format is complex, as it must balance multiple, often conflicting, objectives.

A fundamental challenge in sparse tensor computations is determining how to store, group, and organize the nonzero elements to 1) reduce memory storage, 2) improve data locality, 3) increase parallelism, and 4) decrease workload imbal-

ance and synchronization overhead. Since tensor algorithms perform computations along different mode (i.e., dimension) orientations, practical sparse tensor formats must either be mode-agnostic or, if mode-specific, maintain multiple representations to ensure consistent performance and scalability across all modes.

To tackle this problem, researchers have proposed many sparse tensor formats [3, 6, 90, 87, 57, 79, 52, 53, 70, 69], which can be classified based on their encoding of the multi-dimensional coordinates into list-, tree-, and block-based formats [17] (Table 2.2). In this section, we review and analyze these diverse sparse tensor formats, examining their underlying principles, strengths, and limitations in addressing the challenges of sparse tensor computations.

It is important to note that these sparse tensor format categories are not strictly mutually exclusive. In practice, hybrid approaches are often employed that combine elements from different format types. For example, a format might use a block-based structure for the highest level of the tensor, but employ list-based format for lower levels.

2.3.1 List-based Formats

List-based tensor representations, such as the simple coordinate (COO) format, explicitly store the value of the nonzero elements along with their coordinates (i.e., the indices of all dimensions). Therefore, they are agnostic to the different mode orientations of tensor algorithms and, as a result, they remain the de facto sparse tensor storage [17] in many libraries (e.g., Tensor Toolbox [3], Tensorflow [1], and Tensorlab [103]). Moreover, the COO format closely mirrors common file formats for storing sparse tensors, such as the Matrix Market exchange format [8] and the FROSTT sparse tensor format [85], minimizing preprocessing costs when loading

data. However, list-based COO formats are typically ordered for a single mode, which can lead to various inefficiencies in multi-dimensional operations for other modes, and is often prone to significant synchronization overhead when resolving update/write conflicts across threads [57].

2.3.1.1 Flagged-COO (F-COO)

F-COO [63] is a variant of the standard COO format designed to optimize sparse tensor operations, particularly on GPUs. F-COO extends the COO format by adding two flag arrays: start-flag and bit-flag. These flags enable unified computations for sparse tensor operations and facilitate the application of efficient sparse matrix algorithms like segmented scan. The flagging system in F-COO captures the changes in the computation pattern during sparse tensor operations, such as switching to a new fiber or slice, or transitioning from a dense operation to a sparse mode. This approach eliminates the need for tensor unfolding while requiring less storage compared to formats used in previous tensor literature. By enabling the use of segmented scan algorithms, F-COO improves data reuse and parallelizability on GPUs. The format allows for efficient parallel processing on sparse tensors, enabling simultaneous independent operations on multiple segments of the data. The performance of the proposed unified approach is demonstrated for tensor-based kernels such as the Sparse MTTKRP and the Sparse Tensor-Times-Matrix Multiply (SpTTM) that are widely used in tensor decomposition algorithms.

2.3.2 Tree-based Formats

Several sparse tensor formats use tree-based data structures to extend traditional compressed matrix formats, such as the compressed sparse row (CSR) or col-

umn (CSC) format, to higher-order tensors.

2.3.2.1 Compressed Sparse Fiber (CSF)

The most popular example of these storage representations is the compressed sparse fiber (CSF) format [90], which uses multiple arrays of index pointers to compress the multi-dimensional indices of nonzero elements. While CSF-based formats can reduce memory traffic, they are mode-specific, i.e., they are oriented to favor a specific order of tensor modes. As the tensor order increases, these mode-specific formats require excessive memory to store multiple tensor copies [90, 88] and/or different code implementations are necessary for computing along distinct modes [87]. Moreover, such a compressed, coarse-grained format can suffer from significant workload imbalance and limited scalability, especially in irregularly-shaped tensors with short modes [52].

B-CSF (Balanced Compressed Sparse Fiber) is a sparse tensor format designed to optimize GPU-based tensor operations, particularly MTTKRP [70]. It addresses the limitations of standard CSF by employing a hybrid approach that combines elements of COO, Compressed Slice (CSL), and CSF formats. The key idea behind B-CSF is to adapt the storage format based on the local sparsity pattern within the tensor, enabling fine-grained tensor partitioning. For slices containing only a single element, B-CSF uses the COO format for its simplicity. When a slice has multiple fibers but each fiber contains just one element, it applies the CSL format to compress the slice efficiently. For all other cases, B-CSF adopts the CSF format, leveraging its hierarchical structure. This adaptive strategy allows B-CSF to achieve better load balancing across GPU threads by creating more balanced sub-trees. The fine-grained partitioning also enhances memory efficiency on GPUs by optimizing memory access patterns.

2.3.2.2 Mixed-mode CSF (MM-CSF)

MM-CSF (Mixed-Mode Compressed Sparse Fiber) is a sparse tensor format that partitions nonzero elements into disjoint sections, each compressed along a different mode [69]. Unlike standard CSF, which compresses along a single mode, MM-CSF adapts to varying sparsity patterns by employing different compression strategies for different parts of the tensor. This approach leads to improved compression and better exploitation of the tensor’s structure across multiple modes, resulting in a reduced memory footprint, particularly for tensors with mixed sparsity patterns. Additionally, MM-CSF enhances load balancing in parallel computing environments and demonstrates superior performance in operations like MTTKRP, especially for tensors with irregular sparsity patterns.

Designed to be GPU-friendly, MM-CSF facilitates efficient memory access patterns that are crucial for GPU performance. It allows for improved parallelization and better utilization of GPU cores while achieving effective load balancing across GPU threads. Furthermore, by optimizing the storage format, MM-CSF reduces the amount of data transferred between CPU and GPU memory, addressing a common bottleneck in GPU computations. While MM-CSF offers greater flexibility than single-mode CSF, it is not fully mode-agnostic, as partitioning and mode selection still depend on the tensor’s specific sparsity pattern. Benchmarks demonstrate that MM-CSF significantly improves performance for tensor operations on GPU architectures compared to other formats.

2.3.3 Block-based Formats

2.3.3.1 Hierarchical COO (HiCOO)

HiCOO is a sparse tensor storage method that aims to improve upon the standard COO format by introducing a hierarchical structure [52]. The key idea behind HiCOO is to partition each dimension of the tensor into blocks, similar to the Compressed Sparse Blocks (CSB) format [12]. However, unlike CSB which stores blocks in a dense array, HiCOO employs a sparse storage pattern for these blocks, optimizing for tensors with varying sparsity distributions.

Such block formulation significantly improves both index storage efficiency and performance of tensor operations. By partitioning each dimension of the tensor into blocks, HiCOO reduces the number of bits required to represent element indices within a block as it stores block coordinates and compressed element offsets within blocks. This leads to substantial index compression, especially for large tensors with high dimensionality. Performance enhancements are also shown from improved data locality and reduced index manipulation overhead. As elements within a block are stored contiguously, the block structure allows for more efficient memory access patterns which improves cache utilization during tensor operations. Additionally, the Z-Morton ordering of blocks further enhances spatial locality across dimensions. When performing index calculations, HiCOO can operate on block indices and then compute element indices using simple offset calculations within blocks. This reduces the complexity of index manipulations compared to COO format, leading to faster execution of tensor operations like MTTKRP (Matricized Tensor Times Khatri-Rao Product). The block structure also facilitates parallelization, as blocks can be processed independently. However, HiCOO’s effectiveness can be limited by the characteristics of the target tensors, such as their shape, density, and data

distribution. The format’s performance is also sensitive to the choice of block size, which may require tuning for optimal results.

2.3.3.2 Reordered HiCOO

The work by [53] proposes two heuristic tensor reordering algorithms to improve data locality for the original HiCOO. It employs BFS-MCS, which uses a breadth-first search approach based on maximum cardinality search, and Lexi-Order, which extends doubly lexical ordering from matrices to tensors. These reordering schemes aim to pack nonzeros closer together, resulting in denser blocks in the HiCOO representation. This can lead to fewer total blocks, more nonzeros per block, and better cache locality during operations such as MTTKRP.

2.3.4 Linearized Formats

Linearized tensor formats represent a novel approach to sparse tensor storage and computation that aims to address some of the limitations of traditional formats like COO and CSF. These formats attempt to map the multi-dimensional tensor space into a linear space, effectively transforming the sparse tensor into a multiple or a single one-dimensional structure. This linearization process preserves spatial locality of non-zero elements in the original tensor, which can lead to improved cache utilization and reduced irregular memory access patterns.

The work in [29] introduces a linearized coordinate (LCO) sparse tensor format where each non-zero element’s index values for each mode are compressed into a single integer called a linearized index value (LIV). This compression not only reduces memory usage but also significantly improves computational efficiency. Operations such as sorting and searching benefit from this structure due to only requir-

ing a single integer comparison per element, as opposed to up to N comparisons in COO formats. This reduction in comparison operations translates to faster sorting and rearrangement procedures, which are fundamental to tensor arithmetic.

The LCO format also supports LIV shuffle operations, which rearrange the lexicographical order of the tensor to accommodate specific mode orderings required by different tensor operations. While this capability allows LCO to adapt to various mode orientations, it can be considered a mode-specific aspect of the representation, as it necessitates an LIV shuffle to optimize the sparse tensor format for operations along a particular mode.

The ALTO (Adaptive Linearized Tensor Order) format [32, 48] is a novel approach for storing and processing sparse tensors, designed to address the challenges posed by irregular shapes and data distributions in high-dimensional sparse data. ALTO maps the multi-dimensional coordinates of nonzero elements in a sparse tensor to positions on a compact line using an adaptive bit encoding scheme. This scheme employs variable-sized bit groups to efficiently handle tensors of arbitrary dimensionality, adapting to the unequal cardinalities of different modes. By using a single representation for all modes, ALTO eliminates the need for multiple tensor copies or mode-specific implementations.

One of ALTO’s key features is its adaptive synchronization mechanism, which efficiently handles conflicting updates across threads based on the data reuse patterns of the tensor. This approach, combined with ALTO’s linearized storage, leads to improved data locality, reduced storage requirements, and better workload balancing. The format is versatile across a wide range of tensor shapes and sizes, enabling efficient execution of key tensor operations such as MTTKRP. Additionally, ALTO typically requires less time for format conversion compared to other compressed formats, making it a promising solution for high-performance sparse tensor

computations.

ALTO-stream [92] extends the original ALTO work by introducing dynamic tensor linearization and dynamic time slicing to efficiently handle streaming tensor data. The dynamic tensor linearization process efficiently encodes incoming tensor data into a compact representation on-the-fly. It analyzes new nonzero elements to identify unique objects or entities, expanding factor matrices as needed. Additionally, it applies spatial sketching to remap nonzero elements into a more compact multi-dimensional space, significantly reducing memory usage and improving data locality.

The framework also incorporates dynamic time slicing, which uses variable-width time slices to balance the frequency of factor updates and computing resource utilization. This adaptive processing approach dynamically adjusts to the characteristics of each time slice, ensuring efficient resource utilization, especially for sparse data. By combining these techniques, ALTO-stream achieves substantial performance improvements over state-of-the-art methods, particularly in handling the challenges of extreme sparsity and high sparsity variance in streaming tensors.

The BLCO format [67] extends the ALTO format to work efficiently on massively parallel architectures like GPUs. It employs adaptive blocking to partition the tensor into coarse-grained blocks that fit within GPU memory constraints, enabling the processing of out-of-memory tensors and efficient utilization of GPU resources. It re-encodes linearized indices to allow fast de-linearization using bitwise operations supported by GPUs, rather than relying on expensive bit-level scatter/gather operations. The blocked structure of BLCO allows for the processing of tensors larger than GPU memory, making it suitable for handling large-scale sparse tensors that exceed the capacity of GPU memory.

Characteristic	CP-Decomposition	Tucker Decomposition	Tensor-Train Decomposition
Full Name	Canonical Polyadic Decomposition	Tucker Decomposition	Tensor-Train Decomposition
Also Known As	CANDECOMP/PARAFAC	Higher-Order SVD (HOSVD)	Matrix Product State (MPS)
Decomposition Form	Sum of rank-1 tensors	Core tensor with factor matrices	Chain of 3rd-order tensors
Number of Parameters	$O(NRD)$	$O(NR + R^D)$	$O(DNR^2)$
Main Algorithms	Alternating Least Squares (ALS), Gradient-based methods	Higher-Order Orthogonal Iteration (HOOI), Truncated HOSVD	TT-SVD, Cross-approximation
Rank Definition	Single rank R	Multi-linear rank $(R_1, R_2, \dots, R_D)$	TT-ranks $(r_1, r_2, \dots, r_{D-1})$
Scalability	Moderate	Moderate	High (linear in D)
Interpretability	High	Moderate	Moderate
Main Applications	Latent factor analysis, Feature extraction	Dimensionality reduction, Subspace learning	High-dimensional data/model compression, Solving High-dimensional linear systems

Table 2.3. Overview comparison of CP-Decomposition, Tucker Decomposition, and Tensor-Train Decomposition involves analyzing their performance and suitability based on certain parameters. N represents the length of each mode or dimension, R denotes the rank, and D is the number of dimensions. Each decomposition method has unique characteristics and computational complexities influenced by these parameters, which affect their applicability to different types of tensor data.

2.4 Tensor Factorization

Tensor factorization (or decomposition) is a powerful technique for analyzing and extracting meaningful information from high-dimensional tensors [44, 105, 97, 46, 83, 77, 60]. The primary objective of tensor factorization is to represent a high-dimensional, complex tensor as a combination of simpler, lower-dimensional components, effectively creating a parametric model of the data. This approach allows for dimensionality reduction, feature extraction, and the discovery of latent structures within multidimensional datasets.

Several types of tensor decomposition methods exist, each with unique strengths and applications. The three most prominent types are CANDECOMP/PARAFAC (CP) Decomposition, Tucker Decomposition, and Tensor-Train Decomposition.

Canonical polyadic decomposition (CPD) represents a tensor as a sum of rank-one tensors, also known as canonical polyadic (CP) tensors. This method is particularly useful for latent parameter estimation in machine learning models, signal processing and source separation, as well as chemometrics and spectral analysis. Tucker decomposition, also known as higher-order singular value decomposi-

tion (HOSVD), factorizes a tensor into a core tensor multiplied by a matrix along each mode. Its applications include subspace estimation and dimensionality reduction, data compression and feature extraction, and image and video processing [15, 6, 40, 88, 47]. Tensor-train decomposition represents a tensor as a chain of lower-dimensional tensors, making it especially suitable for high-order tensors [75, 71, 45, 54, 74]. This method finds applications in quantum many-body physics simulations [22], large-scale numerical computations, and machine learning model compression [83, 72, 2, 76, 27]. Table 2.3 presents a comparative overview of the three primary tensor decomposition methods: CP, Tucker, and Tensor-Train.

For the scope of this dissertation, we will focus solely on canonical polyadic decomposition (CPD). In the following subsection, we will explore the fundamental algorithmic approaches for CPD and their adaptations for sparse tensor datasets. Additionally, we will examine the streaming tensor case for CPD, which is particularly relevant to the work presented in this dissertation.

2.4.1 Canonical polyadic (CP) decomposition

Canonical polyadic decomposition (CPD) approximates a mode- N tensor as a sum of K rank-1 tensors, each formed by the outer product of N vectors. The parameter K , referred to as the rank of the decomposition, is typically a small integer, usually in the range of 10 to 100. This approach is known as low-rank tensor factorization. CPD is a low-rank approximation method analogous to the truncated singular value decomposition (SVD) for matrices, extending the concept of matrix factorization to higher-dimensional tensor data.

The K vectors for each mode, used to form the rank-1 tensors, constitute the columns of the factor matrices. There is one factor matrix for each of the N modes.

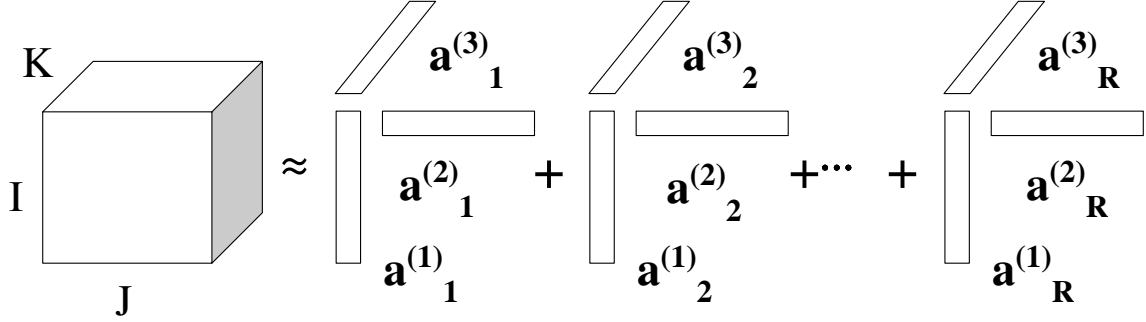


Figure 2.2. Rank- R CPD of a third-order tensor. The factor matrix $\mathbf{A}^{(1)}$ consists of vectors $\mathbf{a}_1^{(1)}, \mathbf{a}_2^{(1)}, \dots, \mathbf{a}_R^{(1)}$.

We denote the factor matrix for the n^{th} mode as $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K}$, where I_n represents the dimension size for that mode. Figure 2.2 illustrates this decomposition structure.

CPD can be represented by the following optimization problem:

$$\underset{\{\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K}\}}{\text{minimize}} \quad \frac{1}{2} \left\| \mathcal{X} - \llbracket \mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)} \rrbracket \right\|^2 \quad (2.2)$$

where we use the notation $\llbracket \cdot \rrbracket$ to represent the Kruskal tensor, which is the reconstructed tensor using the factor matrices $\mathbf{A}_{(n)}$.

$$\llbracket \mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)} \rrbracket = \sum_{k=1}^K \mathbf{a}_k^{(1)} \circ \dots \circ \mathbf{a}_k^{(N)}. \quad (2.3)$$

Here, $\circ$ represents the outer product of vectors, and $\mathbf{a}_k^{(n)}$ is the k -th column of $\mathbf{A}^{(n)}$ (Figure 2.2).

This formulation of the CPD optimization problem is inherently non-convex as the objective function involves products of factor matrices, leading to a non-convex landscape with multiple local minima [46]. This non-convexity makes it challenging to find a global optimum using traditional convex optimization techniques. To address this challenge, a Block Coordinate Descent (BCD) approach, specifically the CP Alternating Least Squares (CP-ALS) algorithm [14, 30], is commonly employed.

Algorithm 1 CP-ALS

Input: An N -order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$, randomly initialized dense factor matrices $\mathbf{A}^{(1)} \in \mathbb{R}^{I_1 \times R}$, $\mathbf{A}^{(2)} \in \mathbb{R}^{I_2 \times R}$, $\dots$, $\mathbf{A}^{(N)} \in \mathbb{R}^{I_N \times R}$.
Output: Updated factor matrices that approximate $\mathcal{X}$.

- 1: **repeat**
- 2: **for** $n = 1, \dots, N$ **do**
- 3: $\mathbf{V} \leftarrow \mathbf{A}^{(1)T} \mathbf{A}^{(1)} * \dots * \mathbf{A}^{(n-1)T} \mathbf{A}^{(n-1)} * \mathbf{A}^{(n+1)T} \mathbf{A}^{(n+1)} * \dots * \mathbf{A}^{(N)T} \mathbf{A}^{(N)}$
- 4: $\mathbf{M} \leftarrow \mathbf{X}_{(n)}(\mathbf{A}^{(N)} \odot \dots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \dots \odot \mathbf{A}^{(1)})$
- 5: $\mathbf{A}^{(n)} \leftarrow \mathbf{M} \mathbf{V}^\dagger$ $\triangleright \dagger$ denotes the pseudo-inverse
- 6: **end for**
- 7: **until** fit ceases to improve or maximum # of iterations reached
- 8: **return** $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}$

CP-ALS iteratively optimizes each factor matrix while keeping the others fixed, effectively breaking down the non-convex problem into a series of convex subproblems. This approach alternates between updating each factor matrix using least squares, which is convex and easily solvable. While CP-ALS does not guarantee convergence to a global optimum due to the problem's non-convexity, it often provides satisfactory results in practice and is widely used for its simplicity and effectiveness in handling the CPD optimization problem. The pseudocode for CP-ALS is illustrated in Algorithm 1.

In recent years, the dynamic nature of data has necessitated the development of online or streaming tensor factorization algorithms, such as CP-Stream, that can adapt to incremental updates in the data. Unlike traditional batch algorithms, these online algorithms are designed to process and update tensor decompositions in a real-time fashion. This is particularly useful in scenarios where the tensor data is constantly evolving and re-computation of the entire decomposition is computationally infeasible.

CHAPTER 3

ROW-SPARSE CP-STREAM

This chapter includes material previously published at the 2021 International Parallel and Distributed Processing Symposium (IPDPS) in Section 3.3 and Section 3.4. This work was a collaborative effort involving the University of Oregon, the Parallel Computing Lab at Intel, and researchers from Meta, Google, and Microsoft. As the first author, I was responsible for the implementation, experimental design, and execution, as well as drafting the majority of the manuscript. Co-authors Dr. Patrick Flick, Dr. Xing Liu, Dr. Shaden Smith, Dr. Fabio Checconi, Dr. Fabrizio Petrini, and Dr. Jee Choi provided technical guidance and assisted with reviewing and editing the text.

3.1 Problem Statement

In this chapter, we investigate a variant of sparse tensor factorization known as streaming sparse tensor factorization. We identify key challenges stemming from the hyper-sparse nature of these streaming tensors. To overcome these challenges, we introduce a novel algorithm. This work directly addresses our first research question: **RQ1**: How can we optimize streaming sparse tensor factorization throughput by exploiting the hyper-sparse nature of the streaming sparse tensors?

3.2 Tensor Factorization for Streaming Sparse Tensors

In previous chapters, we have addressed sparse tensors and their factorization algorithms (CP-ALS). In this section, we further extend our discussion to the realm of streaming sparse tensors and their factorization algorithms.

Table 3.1. The target streaming sparse tensors, ordered by the number of nonzero elements. The temporal mode is underlined.

Tensor	Dimensions	NNZ	Density
NIPS	$2.5K \times 2.9K \times 14K \times \underline{17}$	3.1M	1.8e−06
Uber	$\underline{183} \times 24 \times 1.1K \times 1.7K$	3.3M	3.8e−04
Chicago	$\underline{6.2K} \times 24 \times 77 \times 32$	5.3M	1.5e−02
LANL	$\underline{3.8K} \times 11.2K \times 8.7K \times 75.1K \times 9$	69.1M	2.8e−10
Flickr	$319.7K \times 28.2M \times 1.6M \times \underline{731}$	112.9M	1.1e−14
Delicious	$532.9K \times 17.3M \times 2.5M \times \underline{1.4K}$	140.1M	4.3e−15
Patents	$\underline{46} \times 239.2K \times 239.2K$	3.6B	1.4e−03

3.2.1 Streaming Sparse Tensors

Revisiting Table 2.1, we observe that several of the real-world sparse tensors can be considered as streaming tensors. Meaning that a specific mode of the tensor represents the temporal mode, and can be assumed to be arriving at each time slice. We show in Table 3.1, that several of the real-world sparse tensors can be considered as streaming tensors. Meaning that a specific mode of the tensor represents the temporal mode, and can be assumed to be arriving at each time slice.

3.2.2 Challenges in Streaming Sparse Tensor Factorization

In a streaming setting, new data is continually generated, requiring tensor factorization algorithms that can track the emerging temporal trends by *efficiently* analyzing incoming nonzero elements, and *incrementally* updating the factors to accurately reflect any new information. Specifically, given an infinite source of high-dimensional sparse data, the main goal of streaming tensor factorization (STF) is to minimize the time required to (i) *ingest* new tensor data and (ii) *update* the factors, without having to explicitly construct the complete tensor and compute new factors at every time instance. However, achieving this goal is difficult as streaming sparse tensors demonstrate *drastically different data properties* from traditional

(static) sparse tensors, creating unique challenges in delivering efficient STF on modern processors.

The tensor decomposition problem in a streaming setting can be defined as finding the rank- K CPD of an $(N+1)$ -way tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times T}$, in which N -way tensors arrive over time in T batches. The time T can be potentially unbounded (i.e., $T \rightarrow \infty$). Finding the solution to this problem can be described by the following optimization problem:

$$\underset{\{\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K}\}, \mathbf{S} \in \mathbb{R}^{T \times K}}{\text{minimize}} \quad \frac{1}{2} \left\| \mathcal{Y} - \llbracket \mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}, \mathbf{S} \rrbracket \right\|^2 \quad (3.1)$$

where $\{\mathbf{A}^{(n)}\}$ are the factor matrices for the N -way tensors that are streamed in, and $\mathbf{S}$ is the factor matrix that incorporates the temporal information. The term $\llbracket \mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}, \mathbf{S} \rrbracket$ is an abbreviation of the outer-product formulation for approximating a tensor using its factor matrices, as described above.

$\mathcal{Y}$ can equivalently be modeled as a *sequence* of N -way tensors, $\mathcal{X}_1, \dots, \mathcal{X}_T$, with each $\mathcal{X}_t$ modeled by $\llbracket \mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}; \mathbf{s}_t \rrbracket$, where $\mathbf{s}_t \in \mathbb{R}^K$. This leads to the following optimization problem:

$$\underset{\{\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K}\}, \{\mathbf{s}_t \in \mathbb{R}^K\}}{\text{minimize}} \quad \sum_{t=1}^T \frac{1}{2} \left\| \mathcal{X}_t - \llbracket \mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}; \mathbf{s}_t \rrbracket \right\|^2 \quad (3.2)$$

With this formulation, an algorithm that estimates the solution by considering *one tensor sample $\mathcal{X}_t$ at a time* can be defined. We first solve for $\mathbf{s}_t$ which has a closed-form solution that depends only on factor matrices from the previous time slice. Once we obtain $\mathbf{s}_t$, we then update the factor matrices for the current time slice.

Ideally, updating the factor matrices would be done by taking all the historical data $\mathcal{X}_1, \dots, \mathcal{X}_t$ into consideration; however, in a streaming setting, the number of tensors can grow very large, making this infeasible.

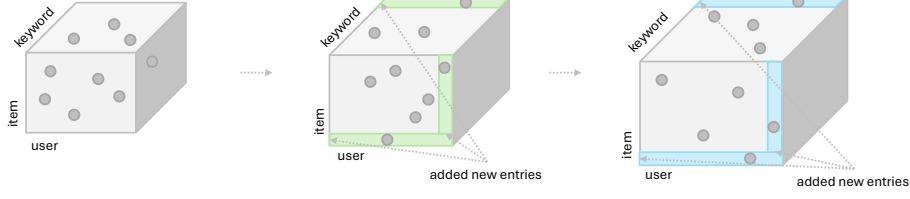


Figure 3.1. Illustration of streaming sparse tensors. As new incoming data arrives, the tensor is constantly being updated and the dimensions of the tensor increases by newly added entries.

3.2.3 CP-stream

Instead of taking into account all historical data $\mathcal{X}_1, \dots, \mathcal{X}_t$, *CP-stream* [86] adopts the approach by Vandescapelle et al. [101], and minimizes an *approximate loss* by replacing $\mathcal{X}_1, \dots, \mathcal{X}_t$ with the existing factorization $\llbracket \mathbf{A}_{t-1}^{(1)}, \dots, \mathbf{A}_{t-1}^{(N)}, \mathbf{S}_{t-1} \rrbracket$, where $\mathbf{S}_{t-1} \in \mathbb{R}^{t-1 \times K}$ is the matrix with rows $s_1, \dots, s_{t-1}$.

The reformulated objective can be written as follows:

$$\begin{aligned} \underset{\mathbf{A}^{(n)} \in \mathcal{C}}{\text{minimize}} \quad & \frac{1}{2} \left\| \mathcal{X}_t - \llbracket \mathbf{A}_t^{(1)}, \dots, \mathbf{A}_t^{(N)}; \mathbf{s}_t \rrbracket \right\|^2 + \\ & \sum_{i=1}^{t-1} \frac{\mu^{t-i}}{2} \left\| \llbracket \mathbf{A}_{t-1}^{(1)}, \dots, \mathbf{A}_{t-1}^{(N)}; \mathbf{s}_i \rrbracket - \llbracket \mathbf{A}_t^{(1)}, \dots, \mathbf{A}_t^{(N)}; \mathbf{s}_t \rrbracket \right\|^2 \end{aligned} \quad (3.3)$$

where $\mu \in [0, 1]$ is the forgetting factor that downweighs the importance of historical data. For problems that require constraints, constraint $\mathcal{C}$ can be applied to the factor matrices $\mathbf{A}^{(n)}$.

Since the objective still requires all historical s_i , the *CP-stream* algorithm further relaxes this requirement by introducing a Gram matrix containing all previous temporal information, $\mathbf{G}_{t-1}$. The overall *CP-stream* algorithm is described in Algorithm 2 and we direct interested readers to [86] for more extensive details.

If we wish to apply constraints to Algorithm 2, constrained optimization kernels such as the ADMM kernel can be used in line 7 instead to apply constraints to the factor matrices, such as non-negativity or sparsity. The inner workings of con-

Algorithm 2 CP-stream

Input: An N -order sparse tensor stream, initial factor matrices $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \dots, \mathbf{A}^{(N)}$

Output: Updated factor matrices $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \dots, \mathbf{A}^{(N)}$ that approximate the tensor stream

```
1: for  $t = 1, \dots, T$  do
2:    $s_t \leftarrow$  least-squares update
3:   repeat
4:     for  $n = 1, \dots, N$  do
5:        $\Phi^{(n)} \leftarrow \left( \bigotimes_{v \neq n}^N \mathbf{A}^{(v)\top} \mathbf{A}^{(v)} \right) \otimes (\mu \mathbf{G}_{t-1} + s_t s_t^\top)$ 
6:        $\Psi^{(n)} \leftarrow \text{MTTKRP} \left( \mathcal{X}_t, \{\mathbf{A}^{(v)}\}, n \right) + \mathbf{A}_{t-1}^{(n)} \left( \left( \bigotimes_{v \neq n}^N \mathbf{A}_{t-1}^{(v)\top} \mathbf{A}^{(v)} \right) \otimes \mu \mathbf{G}_{t-1} \right)$ 
7:        $\mathbf{A}^{(n)} \leftarrow \Psi^{(n)} (\Phi^{(n)})^{-1}$ 
8:     end for
9:   until convergence
10: end for
11: return  $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}$ 
```

strained optimization kernels will be further explored and analyzed in Chapter 5 and Chapter 6 and for the scope of this chapter we will focus on strictly optimizing unconstrained CP-stream performance.

3.3 Methodology: Row-Sparse CP-stream

In this section, we provide a detailed description for *Row-Sparse CP-stream* (*spCP-stream*) — our new algorithm for calculating streaming tensor decomposition.

3.3.1 Motivation

During our evaluation, we observed that for certain tensors, the non-zero elements were clustered around specific index values for some modes. For example, the *Flickr* dataset [85] shown in Figure 3.2 has $113M$ non-zero elements and the length of mode 2 is $28M$. This translates to $113M$ tags being made on *some* of the

$28M$ total available *images*. Due to the streaming nature of this data, only a small subset of images are being tagged at certain times (e.g., images are never tagged again after the initial tag and upload). This means that for every time slice, the mode 2 index values for every non-zero elements can be limited to a small subset of $28M$ possible index values.

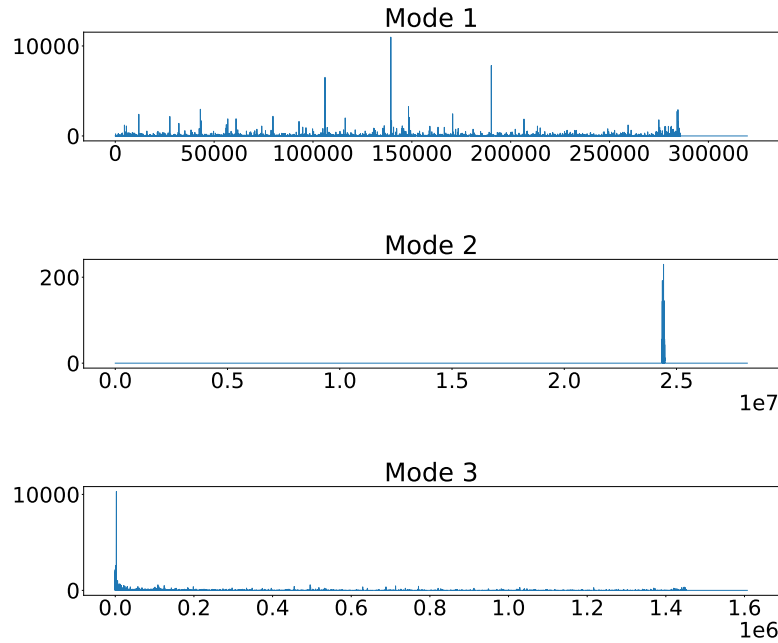


Figure 3.2. Histogram of non-zero elements for the *Flickr* dataset at time slice 500. For mode 2 and 3 specifically, all of the non-zero elements occupy a small subset of index values. In contrast, the elements are more evenly distributed among all index values for mode 1.

To make data access more efficient, we created a new data structure for storing the factor matrices, where we divide the factor matrices into two non-overlapping subsets of rows — one subset with rows that are updated, and the other with rows that are never updated during *MTTKRP* — and update the two subsets individually as required. Propagating this data structure to the rest of the algorithm led to

a new algorithm that greatly reduces the computational complexity of the *overall algorithm*.

3.3.2 spCP-stream Notations

Here are additional notations specific to *spCP-stream*:

1. For a given time slice $\mathcal{X}_t$, the set of mode n index values for every non-zero element is denoted by $\mathbf{nz}^{(n)}$.
2. We use $\mathbf{A}_{nz}^{(n)}$ to denote the matrix formed by “gathering” the rows of the factor matrix $\mathbf{A}^{(n)}$ if the row index is in $\mathbf{nz}^{(n)}$. We also denote this by $\mathbf{A}_{nz}^{(n)} \leftarrow \mathbf{A}^{(n)} [\mathbf{nz}^{(n)}]$.
3. We use $\mathbf{A}_z^{(n)}$ to denote the matrix formed by taking all remaining rows of $\mathbf{A}^{(n)}$. That is, $\mathbf{A}_z^{(n)} \leftarrow \mathbf{A}^{(n)} [\mathbf{z}^{(n)}]$, where $Z \leftarrow \{1, \dots, I_n\}$, and $\mathbf{z}^{(n)} \leftarrow Z \setminus \mathbf{nz}^{(n)}$.
4. We use $\oplus$ to denote forming a new matrix by “gathering” the rows from $\mathbf{A}_z^{(n)}$ and $\mathbf{A}_{nz}^{(n)}$. That is, $\mathbf{A}_{nz}^{(n)} \oplus \mathbf{A}_z^{(n)} \equiv \mathbf{A}^{(n)} [\mathbf{nz}^{(n)} \cup \mathbf{z}^{(n)}]$.
5. *Sparse MTTKRP* (*spMTTKRP*) is our new *MTTKRP* implementation that operates *only* over the non-zero rows (i.e., over $\mathbf{A}_{nz}^{(n)}$)

3.3.3 spCP-stream Algorithm

To fully utilize the benefit of separating the factor matrix into two subsets of non-zero rows (i.e., $\mathbf{A}_z^{(n)}$ and $\mathbf{A}_{nz}^{(n)}$), we need to formulate an *end-to-end computation scheme* that operates over these matrices.

If we collapse the inner-most loop in Algorithm 2 (i.e., combine lines 5 through 7), then we have

$$\begin{aligned} \mathbf{A}^{(n)} = & \left(MTTKRP \left(\mathcal{X}_t, \{\mathbf{A}^{(v)}\} \right) + \right. \\ & \left. \mathbf{A}_{t-1}^{(n)} \left(\left(\bigotimes_{v \neq n} \mathbf{A}_{t-1}^{(v)T} \mathbf{A}^{(v)} \right) \bigotimes \mu \mathbf{G}_{t-1} \right) \right) \cdot \\ & \left(\left(\bigotimes_{v \neq n}^N \mathbf{A}^{(v)T} \mathbf{A}^{(v)} \right) \bigotimes (\mu \mathbf{G}_{t-1} + s_t s_t^T) \right)^{-1} \end{aligned} \quad (3.4)$$

We now separate this out to two parts — one for computing $\mathbf{A}_{nz}^{(n)}$ and the other for computing $\mathbf{A}_z^{(n)}$. For $\mathbf{A}_z^{(n)}$, $MTTKRP$ makes no contribution, and thus can be eliminated:

$$\begin{aligned} \mathbf{A}_z^{(n)} = & \mathbf{A}_{z,t-1}^{(n)} \underbrace{\left(\left(\bigotimes_{v \neq n} \mathbf{A}_{t-1}^{(v)T} \mathbf{A}^{(v)} \right) \bigotimes \mu \mathbf{G}_{t-1} \right)}_{\mathbf{Q}^{(n)}} \cdot \\ & \underbrace{\left(\left(\bigotimes_{v \neq n}^N \mathbf{A}^{(v)T} \mathbf{A}^{(v)} \right) \bigotimes (\mu \mathbf{G}_{t-1} + s_t s_t^T) \right)^{-1}}_{(\Phi^{(n)})^{-1}} \end{aligned} \quad (3.5)$$

Simplifying this equation, we end up with:

$$\mathbf{A}_z^{(n)} = \mathbf{A}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} (\Phi^{(n)})^{-1} \quad (3.6)$$

As for $\mathbf{A}_{nz}^{(n)}$, we can equivalently simplify the equation as follows:

$$\mathbf{A}_{nz}^{(n)} = \left(spMTTKRP \left(\mathcal{X}_t, \{\mathbf{A}_{nz}^{(v)}\} \right) + \mathbf{A}_{nz,t-1}^{(n)} \mathbf{Q}^{(n)} \right) (\Phi^{(n)})^{-1} \quad (3.7)$$

where $spMTTKRP$ is our new $MTTKRP$ implementation that operates over only the non-zero rows. Notice that both $\mathbf{Q}^{(n)}$ and $\Phi^{(n)}$ are small $K \times K$ Gram matrices, and $\Phi^{(n)}$ is positive definite.

Also, we define:

$$\mathbf{C}^{(n)} = \mathbf{A}^{(n)T} \mathbf{A}^{(n)} \quad (3.8)$$

$$\mathbf{H}^{(n)} = \mathbf{A}_{t-1}^{(n)T} \mathbf{A}^{(n)} \quad (3.9)$$

$\mathbf{C}^{(n)}$ and $\mathbf{H}^{(n)}$ are also $K \times K$ Gram matrices, and can be written as the sum of the Gram matrices of $\mathbf{A}_z^{(n)}$ and $\mathbf{A}_{nz}^{(n)}$. For example,

$$\begin{aligned} \mathbf{C}^{(n)} &= \mathbf{A}^{(n)T} \mathbf{A}^{(n)} \\ &= \underbrace{\mathbf{A}_z^{(n)T} \mathbf{A}_z^{(n)}}_{\mathbf{C}_z^{(n)}} + \underbrace{\mathbf{A}_{nz}^{(n)T} \mathbf{A}_{nz}^{(n)}}_{\mathbf{C}_{nz}^{(n)}} \end{aligned} \quad (3.10)$$

$H^{(n)}$ decomposes in the same way.

Notice that from Equation 3.6, we have

$$\begin{aligned} \mathbf{A}_z^{(n)T} \mathbf{A}_z^{(n)} &= \left(\mathbf{A}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} (\Phi^{(n)})^{-1} \right)^T \mathbf{A}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} (\Phi^{(n)})^{-1} \\ &= (\Phi^{(n)})^{-T} \mathbf{Q}^{(n)T} \mathbf{A}_{z,t-1}^{(n)T} \mathbf{A}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} (\Phi^{(n)})^{-1} \\ &= (\Phi^{(n)})^{-T} \mathbf{Q}^{(n)T} \mathbf{C}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} (\Phi^{(n)})^{-1} \end{aligned} \quad (3.11)$$

Putting it all together, we can calculate $\mathbf{C}^{(n)}$ (Equation 3.10) by adding the results of Equation 3.11 to the Gram matrix of the result of Equation 3.7 (i.e., $\mathbf{A}_{nz}^{(n)T} \mathbf{A}_{nz}^{(n)}$).

Similarly, we can calculate $\mathbf{H}^{(n)}$ by calculating

$$\mathbf{H}_{nz}^{(n)} = \underbrace{\mathbf{A}_{nz,t-1}^{(n)T}}_{\text{from previous time slice}} \underbrace{\mathbf{A}_{nz}^{(n)}}_{\text{Equation 3.7}} \quad (3.12)$$

$$\begin{aligned} \mathbf{H}_z^{(n)} &= \mathbf{A}_{z,t-1}^{(n)T} \mathbf{A}_z^{(n)} \\ &= \mathbf{A}_{z,t-1}^{(n)T} \underbrace{\mathbf{A}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} (\Phi^{(n)})^{-1}}_{\text{Equation 3.6}} \\ &= \mathbf{C}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} (\Phi^{(n)})^{-1} \end{aligned} \quad (3.13)$$

Finally, the calculation for $\mathbf{Q}^{(n)}$ and $\Phi^{(n)}$ reduces to:

$$\begin{aligned}
\mathbf{Q}^{(n)} &= \left(\bigotimes_{v \neq n} \mathbf{A}_{t-1}^{(v)T} \mathbf{A}^{(v)} \right) \bigotimes \mu \mathbf{G}_{t-1} \\
&= \left(\bigotimes_{v \neq n} \mathbf{H}^{(v)} \right) \bigotimes \mu \mathbf{G}_{t-1} \\
\Phi^{(n)} &= \left(\bigotimes_{v \neq n}^N \mathbf{A}^{(v)T} \mathbf{A}^{(v)} \right) \bigotimes (\mu \mathbf{G}_{t-1} + s_t s_t^T) \\
&= \left(\bigotimes_{v \neq n}^N \mathbf{C}^{(v)} \right) \bigotimes (\mu \mathbf{G}_{t-1} + s_t s_t^T)
\end{aligned} \tag{3.14}$$

In conclusion, by maintaining and updating the small $K \times K$ Gram matrices $\mathbf{C}^{(n)}$ and $\mathbf{H}^{(n)}$ in the inner loop, we can reduce the amount of computation required for calculating $\mathbf{Q}^{(n)}$, $\Phi^{(n)}$. The overall algorithm is shown in Algorithm 3.

3.3.4 Pre- and Post- Operations

To take advantage of the new formulation described in the previous subsection, some additional *housekeeping* operations are required. For instance, when a new time slice arrives, it is necessary to identify the non-zero rows for every mode in advance. Also, because the inner iteration relies on the previous time slice's factor matrices, $\mathbf{C}_{z,t-1}^{(n)}$ needs to be computed prior to the inner iteration.

Once the inner iteration reaches convergence for all modes, we also need to update the full factor matrices $\mathbf{A}^{(n)}$ based on the final $\mathbf{A}_{nz}^{(n)}$ and $\mathbf{A}_z^{(n)}$. However, these newly introduced operations have minimal impact on the overall execution time, as their overhead can be amortized by the inner iterations.

3.3.5 Convergence Check

At the end of each iteration, the normalized absolute difference between $\mathbf{A}^{(n)}$ and $\mathbf{A}_{t-1}^{(n)}$ needs to be calculated to check for convergence. Since *spCP-stream* elim-

inates the need to construct full factor matrices, we also propose a way to check for convergence using only the Gram matrices $\mathbf{C}^{(n)}$ and $\mathbf{H}^{(n)}$.

The termination criteria at time step t is determined by calculating

$$\delta_t = \sum_{n=1}^N \frac{\left\| \mathbf{A}^{(n)} - \mathbf{A}_{t-1}^{(n)} \right\|_F}{\left\| \mathbf{A}^{(n)} \right\|_F} \quad (3.15)$$

and terminating if $|\delta_t - \delta_{t-1}| < \epsilon$ where ϵ is some tolerance.

Using basic linear algebra properties, we can compute the denominator of Equation 3.15 using $\mathbf{C}^{(n)}$ as follows:

$$\left\| \mathbf{A}^{(n)} \right\|_F^2 = \text{tr} \left(\mathbf{A}^{(n)T} \mathbf{A}^{(n)} \right) = \text{tr}(\mathbf{C}^{(n)}) \quad (3.16)$$

Similarly, the numerator of Equation 3.15 can be computed as follows:

$$\begin{aligned} \left\| \mathbf{A}^{(n)} - \mathbf{A}_{t-1}^{(n)} \right\|_F^2 &= \text{tr} \left(\left(\mathbf{A}^{(n)} - \mathbf{A}_{t-1}^{(n)} \right)^T \left(\mathbf{A}^{(n)} - \mathbf{A}_{t-1}^{(n)} \right) \right) \\ &= \text{tr} \left(\mathbf{A}^{(n)T} \mathbf{A}^{(n)} - \mathbf{A}^{(n)T} \mathbf{A}_{t-1}^{(n)} \right. \\ &\quad \left. - \mathbf{A}_{t-1}^{(n)T} \mathbf{A}^{(n)} + \mathbf{A}_{t-1}^{(n)T} \mathbf{A}_{t-1}^{(n)} \right) \\ &= \text{tr} \left(\mathbf{C}^{(n)} - \mathbf{H}^{(n)T} - \mathbf{H}^{(n)} + \mathbf{C}_{t-1}^{(n)} \right) \\ &= \text{tr} \left(\mathbf{C}^{(n)} \right) + \text{tr} \left(\mathbf{C}_{t-1}^{(n)} \right) - 2 \text{tr} \left(\mathbf{H}^{(n)} \right) \end{aligned} \quad (3.17)$$

This yields an equivalent convergence check rule using only the diagonal elements (i.e., *trace*) of $\mathbf{C}^{(n)}$ and $\mathbf{H}^{(n)}$.

3.4 Performance Evaluation and Analysis

For our evaluation, we use a large-memory node from the University of Oregon's *Talapas* high-performance computing (HPC) cluster. This is a quad-socket Intel E7-4830v4 system with a total of 56 cores and 512GB of main memory.

3.4.1 Datasets and Execution Parameters

Datasets used for our evaluation are shown in Table 3.2. We selected these datasets from the FROSTT tensor repository [85] to have varying number non-zero elements, number of modes, and dimension sizes.

Table 3.2. Datasets used in spCP-stream evaluation. The streaming mode is in bold for each data set.

Data	Dimensions	# non-zeros
<i>Patents</i>	year $\times$ terms $\times$ terms 46 $\times$ 239k $\times$ 239k	3.5B
<i>Flickr</i>	user $\times$ image $\times$ tag $\times$ date 320K $\times$ 28M $\times$ 1.6M $\times$ 731	113M
<i>Uber</i>	date $\times$ hour $\times$ lat. $\times$ long. 183 $\times$ 24 $\times$ 1.1K $\times$ 1.7K	3.3M
<i>NIPS</i>	paper $\times$ author $\times$ word $\times$ year 2.5K $\times$ 2.9K $\times$ 14K $\times$ 7	3.1M

For each dataset, we used a forgetting factor of 0.99 and convergence tolerances of 10^{-5} (*Patents* and *Flickr*) and 10^{-6} (*Uber* and *NIPS*). Additionally, we used a Frobenius norm regularization of 10^{-2} on the streaming mode for stability. We used tensor decomposition ranks of $\{16, 32, 64, 128\}$ and number of *OpenMP* threads of $\{1, 7, 14, 28, 56\}$. Lastly, for each decomposition run, random values were used to initialize the factor matrices.

3.4.2 Execution Time, Fit Error, and Convergence Property

The execution times presented in the rest of this section are *per iteration time*, as the number of iterations to convergence for each experimental run vary due to random initialization. We also use the *minimum* per iteration execution time from 10 trials, as the *thread contention* in *MTTKRP* for the baseline implementation

causes a large standard deviation in its execution time. However, the minimum is a more *conservative* measure of the speedup achieved by our optimized kernels and algorithm — if we use the average or the median, the speedup achieved by our work is *significantly* higher across all datasets, as our work shows more consistent execution times. Our work demonstrates similar fit error and convergence properties as the original *CP-stream* algorithm.

3.4.3 Performance Analysis of Unconstrained CP-stream

In many real-world problems, applying constraints to the factor matrices is unnecessary, and the *non-constrained CP-stream* variant can be used, where *NNLS-update* is replaced with a direct factorization-based solver such as *Cholesky*, offering faster performance than the *constrained* counterpart. In this subsection, we present the performance of our new *spCP-stream* algorithm, and compare it against the **baseline** *non-constrained CP-stream*, and our **optimized** implementation that uses our Hybrid Lock *MTTKRP*.

3.4.3.1 Row-Sparse MTTKRP (spMTTKRP)

First, we demonstrate the effectiveness of dividing the factor matrices into two subsets of non-zero rows (i.e., $\mathbf{A}_{nz}^{(n)}$) and zero-rows (i.e., $\mathbf{A}_z^{(n)}$) on the *MTTKRP* kernel. Our *spMTTKRP* kernel achieves as much as $121.1\times$ speedup against the baseline *MTTKRP* kernel, and as much as $6.4\times$ speedup against our own Hybrid Lock *MTTKRP* kernel. The Hybrid Lock *MTTKRP* kernel is an additional optimization developed as part of this work; however, we omit its details in this chapter. For a comprehensive description, we refer readers to [91].

Much of this speedup comes from more efficient memory access, as the ac-

cesses are spread over a much smaller memory footprint (e.g., accessing 10 rows in a 10-row matrix vs. accessing 10 rows in a 1000-row matrix), leading to fewer TLB misses and better pre-fetching performance.

However, it should be noted that identifying the non-zero rows and creating the required data structures do not come for free. They add an extra overhead to our new formulation, and needs to be amortized effectively. Therefore, we do not present any direct comparison between our *Row-Sparse MTTKRP* kernel and others. Instead, we present results that compare the performance of the overall algorithm, including the overhead.

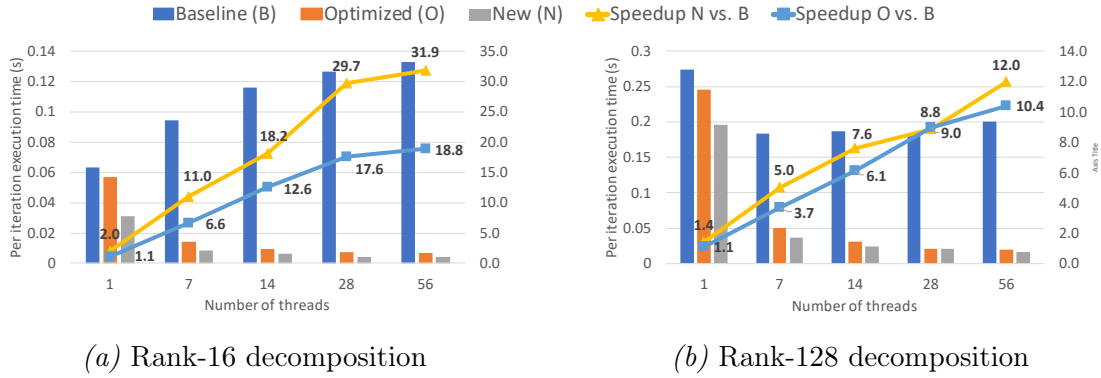


Figure 3.3. Comparison of *spCP-stream* and **optimized non-constrained CP-stream** to the baseline on the *NIPS* dataset.

3.4.3.2 Unconstrained spCP-Stream

In this subsection, we present the performance of our new *spCP-stream* algorithm and compare it against the **baseline** and **optimized non-constrained CP-stream** implementations. Our optimized CP-stream implementation uses our Hybrid Lock *MTTKRP*. Figure 3.3 shows the per iteration execution time for the *NIPS* dataset.

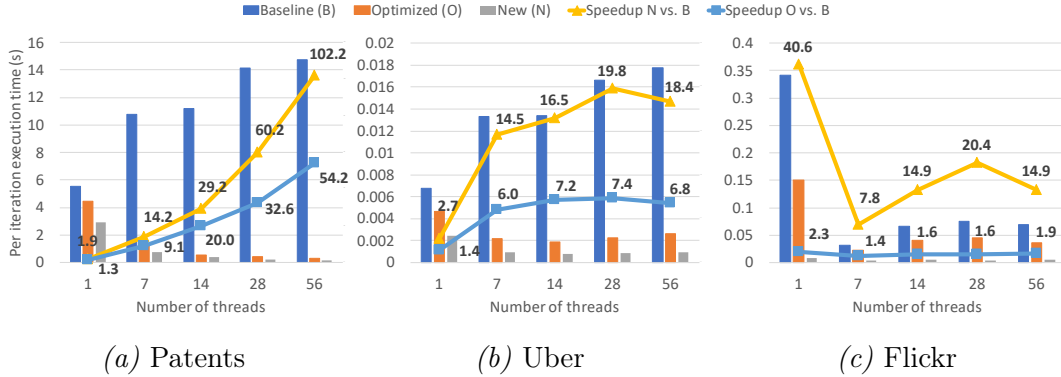


Figure 3.4. Comparison of *spCP-stream* and **optimized non-constrained CP-stream** to the baseline on the *Patents*, *Uber*, and *Flickr* datasets for rank-16 decomposition.

For rank-16 decomposition, our **optimized** implementation achieves $18.8\times$ speedup on 56 threads, whereas our new algorithmic formulation achieves $31.9\times$ speedup. For rank-128 decomposition, the **optimized** implementation and *spCP-stream* achieved $10.4\times$ and $12.0\times$, respectively. The smaller *gap* between the two speedups at higher ranks is due to *spCP-stream* using Gram matrices — as the rank goes up, the computation scales exponentially (i.e., $K \times K$), whereas the original algorithm scales linearly with rank (i.e., $I_n \times K$). We omit the remaining ranks, as they show a similar trend to the rank-16 decomposition, with their speedup sitting between those of rank-16 and rank-128.

Figure 3.4 shows how *spCP-stream* and our **optimized** implementation compare to the baseline on the remaining datasets. First, for the *Patents* dataset, our algorithms scale well up to 28 threads. However, the **baseline** execution time *increases* with the number of threads due to thread contention in *MTTKRP*, leading to $102.2\times$ and $54.2\times$ speedup at 56 threads for *spCP-stream* and **optimized**, respectively.

For the *Uber* dataset, in contrast, the execution time actually goes *up* slightly for all three implementations. This is because the number of non-zero elements and

the dimension sizes are both extremely small, and there is not enough work to keep the system fully occupied.

The *Flickr* dataset shows a *very* different trend. With the other two datasets, *spCP-stream* achieves from $1.5\times$ to $2.7\times$ speedup over our **optimized** implementation. However, with the *Flickr* dataset, this goes up to $5.6\times$ to $17.9\times$. This is due to the extremely sparse nature of the second mode of the *Flickr* dataset (Section 3.3.1), where approximately 99% of the rows are zero rows. This causes our row-sparse *MTTKRP* to perform significantly better than our Hybrid Lock *MTTKRP*, leading to such high speedups for the overall algorithm.

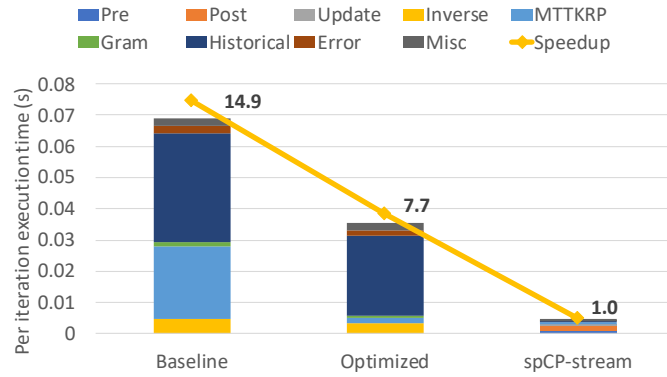


Figure 3.5. Execution time breakdown for the three implementations for the *Flickr* dataset (rank-16 using 56 threads).

3.4.3.3 Execution Time Breakdown

Another reason why *spCP-stream* performs so well on *Flickr* lies in how the *other* computations are also reduced. Figure 3.5 shows a breakdown of where time is spent in each iteration of the three implementations. As it can be seen, our Hybrid Lock *MTTKRP* significantly reduces the time spent in *MTTKRP* (i.e., **baseline** vs. **Optimized**). However, calculating the *Historical* information (Line 5 in Algorithm 2) still takes up a significant amount of time, as it involves multi-

ple matrix-matrix multiplications between extremely large factor matrices (e.g., $28M \times K$ for mode 2). Since we replace this with *Hadamard product between Gram matrices* (i.e., $K \times K$), the computational complexity is greatly reduced, and *spCP-stream* achieves significant speedup against even our own **optimized** implementation. This behavior should occur in any tensors with very large dimension sizes.

3.5 Conclusion

In this chapter, we presented a new algorithm for streaming sparse tensor factorization, called *spCP-stream*. During our study, we discovered that certain real-world tensors have skewed distribution of non-zero elements across different time slices, resulting in these elements updating only a *small subset* of factor matrix rows. To address this issue, we partition the factor matrices into two subsets — one with rows that are updated during *MTTKRP* and those that are not. This resulted in a *new algorithmic formulation of non-constrained CP-stream* that greatly reduces the number of required operations by maintaining the factor matrices in *Gram matrix form*, and computing over these smaller matrices.

This work makes *two key contributions* to improving the performance of streaming tensor decomposition:

- *Optimized constrained CP-stream*, where we use *Blocked & Fused ADMM* and *Hybrid Lock MTTKRP* kernels to improve the performance of the original *constrained CP-stream* algorithm. We achieve up to $47\times$ speedup on our 56-core testbed system on four real-world tensors.
- *A new algorithm for non-constrained CP-stream*, where we partition the factor matrix into two non-overlapping subsets of rows and update them independently. These subsets are stored and computed in Gram matrix form,

which greatly reduces the required computation. Our new algorithm achieves up to $102\times$ over the original *non-constrained CP-stream* algorithm.

Algorithm 3 Row-Sparse CP-stream (spCP-stream)

Input: Streaming tensors $\mathcal{X}_1, \dots, \mathcal{X}_T$; forgetting factor μ

Output: Updated factor matrices $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}$

```

1: Initialize  $\mathbf{A}_0^{(1)}, \dots, \mathbf{A}_0^{(N)}$ 
2:  $\mathbf{G}_0 \leftarrow 0$ 
3: for  $t = 1, \dots, T$  do
4:    $s_t \leftarrow$  closed form update
5:    $\triangleright$  Extract rows of  $\mathbf{C}_{t-1}$  that matches current tensor  $\mathcal{X}_t$ 's nonzero rows
6:   for  $n = 1, \dots, N$  do
7:     if  $t > 1$  then
8:        $\mathbf{nz}^{(n)} \leftarrow \text{nonzero\_slices}(\mathcal{X}_t, n)$ 
9:        $\mathbf{C}_{old} \leftarrow \mathbf{A}^{(n)}[\mathbf{nz}_{t-1}^{(n)} \setminus \mathbf{nz}^{(n)}]^T \mathbf{A}^{(n)}[\mathbf{nz}_{t-1}^{(n)} \setminus \mathbf{nz}^{(n)}]$ 
10:       $\mathbf{C}_{new} \leftarrow \mathbf{A}^{(n)}[\mathbf{nz}^{(n)} \setminus \mathbf{nz}_{t-1}^{(n)}]^T \mathbf{A}^{(n)}[\mathbf{nz}^{(n)} \setminus \mathbf{nz}_{t-1}^{(n)}]$ 
11:       $\mathbf{C}_{z,t-1}^{(n)} \leftarrow \mathbf{C}_{z,t-1}^{(n)} + \mathbf{C}_{old} - \mathbf{C}_{new}$ 
12:    end if
13:  end for
14:  repeat
15:     $\delta \leftarrow 0$ 
16:    for  $n = 1, \dots, N$  do
17:       $\mathbf{Q}^{(n)} \leftarrow \left( \bigoplus_{v \neq n}^* \mathbf{H}^{(v)} \right) \odot \mu \mathbf{G}_{t-1}$ 
18:       $\Phi^{(n)} \leftarrow \left( \bigoplus_{v \neq n}^N \mathbf{C}^{(v)} \right) \odot (\mu \mathbf{G}_{t-1} + s_t s_t^T)$ 
19:       $\mathbf{A}_{nz}^{(n)} \leftarrow \left( \text{spMTTKRP}(\mathcal{X}_t, \{\mathbf{A}_{nz}^{(v)}\}) + \mathbf{A}_{nz,t-1}^{(n)} \mathbf{Q}^{(n)} \right) \cdot (\Phi^{(n)})^{-1}$   $\triangleright$  Solve for non-zero slices
20:       $\mathbf{C}_{nz}^{(n)} \leftarrow \mathbf{A}_{nz}^{(n)T} \mathbf{A}_{nz}^{(n)}$ 
21:       $\mathbf{H}_{nz}^{(n)} \leftarrow \mathbf{A}_{nz,t-1}^{(n)T} \mathbf{A}_{nz}^{(n)}$ 
22:       $\mathbf{H}_z^{(n)} \leftarrow \mathbf{C}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} (\Phi^{(n)})^{-1}$   $\triangleright$  Solve for zero slices
23:       $\mathbf{C}_z^{(n)} \leftarrow (\Phi^{(n)})^{-T} \mathbf{Q}^{(n)T} \mathbf{C}_{z,t-1}^{(n)} \mathbf{Q}^{(n)} (\Phi^{(n)})^{-1}$ 
24:       $\mathbf{C}^{(n)} \leftarrow \mathbf{C}_z^{(n)} + \mathbf{C}_{nz}^{(n)}$   $\triangleright$  Update C and H
25:       $\mathbf{H}^{(n)} \leftarrow \mathbf{H}_z^{(n)} + \mathbf{H}_{nz}^{(n)}$ 
26:       $\triangleright$  Normalize
27:       $\text{normalize}(\mathbf{C}, \mathbf{H})$ 
28:       $\delta_t = \sum_{n=1}^N \sqrt{\frac{\text{tr}(\mathbf{C}^{(n)} - \mathbf{H}^{(n)T} - \mathbf{H}^{(n)} + \mathbf{C}_{t-1}^{(n)})}{\text{tr}(\mathbf{C}^{(n)})}}$ 
29:    end for
30:    until convergence
31:     $\mathbf{A}^{(n)} \leftarrow \mathbf{A}_z^{(n)} \oplus \mathbf{A}_{nz}^{(n)} \forall n \in \{1, \dots, N\}$ 
32:     $\mathbf{G}_t = \mu \mathbf{G}_{t-1} + s_t s_t^T$ 
33:  end for
34: return  $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}$ 

```

CHAPTER 4

STREAMING CPD WITH ADAPTIVE LINEARIZED TENSOR ORDERING

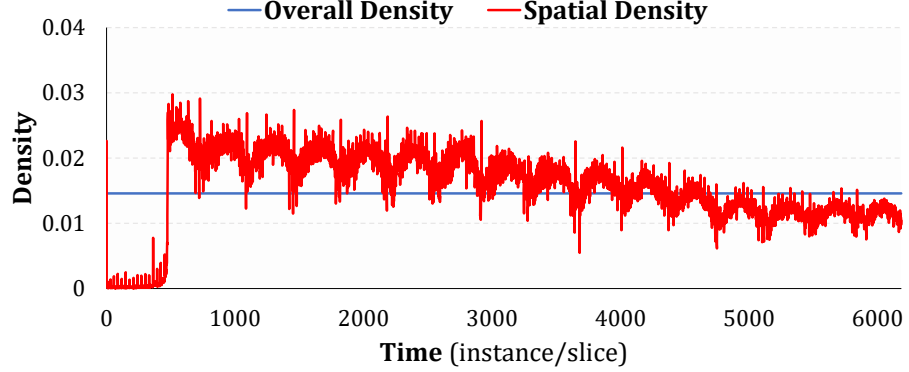
This chapter includes material previously published at the 2023 International Parallel and Distributed Processing Symposium (IPDPS) in Section 4.2, Section 4.3 and Section 4.4. This work was done as a collaborative effort involving the University of Oregon, Parallel Computing Lab at Intel Labs, University of Erlangen-Nurnberg and Laboratory for Physical Sciences. As the first author, I was responsible for the implementation, the majority of the experimentation, as well as the drafting of the manuscript. Co-authors Dr. Ahmed E. Helal, Dr. Fabio Checconi, Ph.D. Candidate Jan Laukemann, Dr. Jesmin Jahan Tithi, Dr. Teresa Ranadive, Dr. Fabrizio Petrini, and Dr. Jee Choi provided technical guidance and assisted with experimentation and reviewing and editing the text.

4.1 Problem Statement

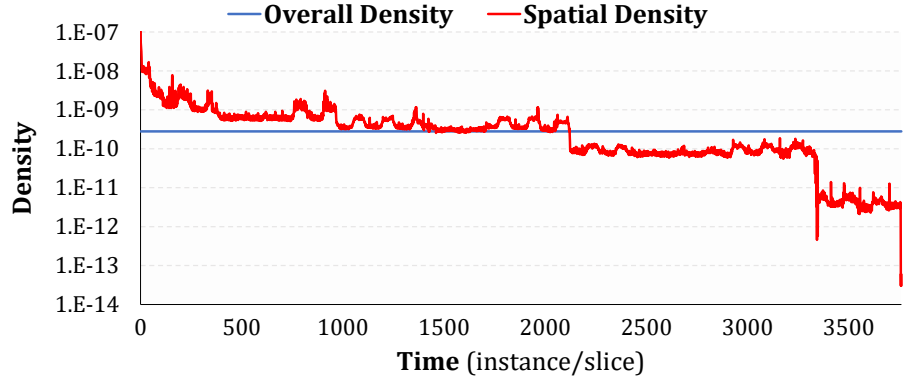
In the previous chapter, we proposed a novel algorithm for streaming sparse tensor factorization that exploits the hyper-sparse nature of the streaming tensors. In this chapter, we address our second research question: **RQ2**: Can we further improve the throughput of streaming sparse tensor factorization utilizing adaptive linearized sparse tensor formats?

4.2 Motivation

In streaming tensors, the tensor’s multi-dimensional space *expands* as interactions evolve with time and more unique objects (or entities) are identified (*e.g.*, when new products are added to reviews). Moreover, in many real-world applications, only small and varying *subsets* of these objects interact at any given time,



(a) Chicago crime statistics data.



(b) LANL network traffic data. Density is in log-scale.

Figure 4.1. Density trend for the (a) Chicago and (b) LANL streaming tensors. The red line shows how the spatial density for each temporal tensor slice changes over time, and the blue line illustrates the overall density after receiving the entire tensor data. Spatial density is the number of nonzero elements in a slice divided by the product of spatial (non-streaming) dimensions, while overall density is the number of nonzero elements across all slices divided by the spatio-temporal volume of the tensor (*i.e.*, the product of all tensor dimensions including the time mode). For both tensors, temporal tensor slices can be *substantially more sparse* than the overall tensor, and *density varies significantly* over time.

and *recent* objects tend to receive/generate more interactions. Hence, the data analysis indicates that streaming tensors can have *significantly higher sparsity* than their corresponding static (non-streaming) tensors, as demonstrated by the difference between the blue and the red lines in Figure 4.1.

Hence, this indicates that streaming tensors can have *significantly higher sparsity* than their corresponding static (non-streaming) tensors, as demonstrated by the difference between the blue and the red lines in Figure 4.1. Specifically, in the Chicago and LANL data sets, the density of time slices can be *three* and *five orders of magnitude* lower than the corresponding static tensor, respectively. Moreover, these time-evolving data sets suffer from *high density variance* across time slices, as indicated by the red spikes in Figure 4.1, showing an overall discrepancy by up to *three* and *six orders of magnitude* for Chicago and LANL, respectively.

To summarize, the extreme sparsity and high sparsity variance of streaming high-dimensional data make it difficult to efficiently execute STF algorithms on modern multi-core processors, as these characteristics lead to random memory access as well as under-utilization of parallel computing resources. Furthermore, due to the prohibitive cost of generating compressed tensor formats on-the-fly [90, 52], the state-of-the-art STF approaches [86, 91] process streaming data in a raw form (*i.e.*, coordinate (COO) format) that fails to capture data locality and suffers from significant synchronization overhead.

4.3 Methodology: ALTO-stream

In this section, we present *ALTO-stream*, a high-performance STF framework that tackles the computational challenges of analyzing time-evolving data and extracting their latent features on-the-fly using *spatial sketching* (4.3.1), *dynamic in-*

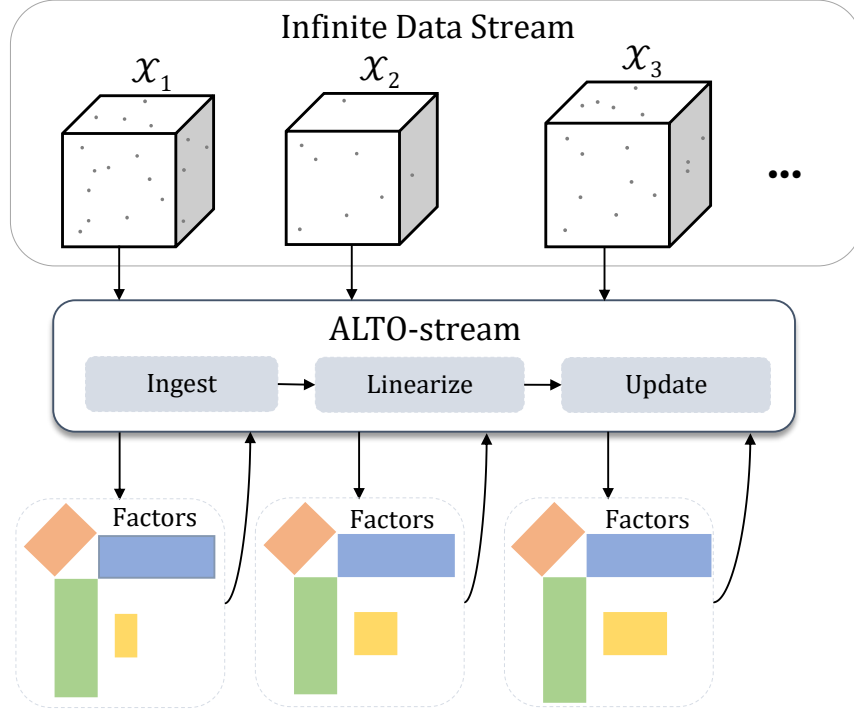


Figure 4.2. Overview of our ALTO-stream framework.

dex linearization (4.3.2), and *dynamic time slicing* (4.3.3). Figure 4.2 shows the overall ALTO-stream framework, which ingests tensor slices and generates updated time-evolving factors.

ALTO-stream first analyzes (*Ingest*) the nonzero elements to identify any new unique objects/entities added to the tensor and expands the length (*i.e.*, number of rows) of the factor matrices accordingly. Notice in Figure 4.2 that the length of the temporal factors (yellow factor matrix) always increases with new time slice computation, while the other factors (e.g., green, blue, and orange factor matrices) grow over time depending on the spatial dimensions of incoming data. For effective dynamic encoding/decoding of highly irregular streaming data, ALTO-stream constructs a spatial sketch of a given tensor slice based on the interactions across objects (entities) for this time period to remap *all* new nonzero elements into a potentially much more compact multi-dimensional space. The nonzero elements are

then linearized and sorted (*Linearize*) for efficient parallel execution of the computational operations (*e.g.*, MTTKRP) required to update the factor matrices (*Update*).

4.3.1 Spatial Sketching

As streaming data evolve over time, new interactions emerge and more unique objects are identified and added to the tensor space as well as the designated factor matrices. However, given the extreme sparsity and sparsity variation across time slices (as shown in Figure 4.1), only a small subset of these objects interact along a specific dimension (mode) at any point in time, leading to highly irregular fiber activation patterns across both spatial and temporal modes. That is, the same fiber can be in different activation states across time slices and neighboring fibers along a spatial mode can have different activation states. For example, in Figure 4.3, the left-most list of nonzero elements (first slice) represents interactions in $20 \times 58 \times 8$ spatial space; however, the number of *interacting* mode-2 index values (or *active fibers*) is only 10 (highlighted green in ❶). Similarly, for mode-1 and mode-3, the number of *interacting* index values are 5 and 4, highlighted blue and orange in ❶, respectively. Due to the irregular spatial distributions of streaming data, it is difficult to find a single subspace that contains all interactions but has a smaller volume. For example, in Figure 4.3, a $20 \times 42 \times 8$ subspace, with boundary coordinates $(0, 16, 0)$ and $(19, 57, 7)$, is the most compact subspace that contains all interactions of the first time slice.

Since the density of high-dimensional data directly impacts the performance of sparse tensor computations [32], we propose to address this problem by using *spatial sketching*, which reduces the tensor space by mapping the original space

to a much smaller *image* space. Traditional sketching [19] reduces both the number of nonzero elements as well as the multi-dimensional space via sampling. However, such a lossy compression can affect the accuracy of STF due to the inexact data approximations. In contrast, by tracking the irregular fiber activation patterns over time, the proposed spatial sketching encodes a series of compact tensor sketches that keep *every* nonzero element (interaction) present in a given time slice but remap these interactions to an image space. Formally, for a tensor slice $\mathcal{X}_t \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_{N-1}}$ with m nonzero elements, a spatial tensor sketch is $\mathcal{X}' \in \mathbb{R}^{I'_1 \times I'_2 \times \dots \times I'_{N-1}}$, where $I'_j \leq I_j \forall j$, such that $\mathcal{X}'$ contains all m nonzero elements in $\mathcal{X}_t$.

To generate $\mathcal{X}'$, we first identify all the active fibers (interacting indices) along every spatial mode in a given time slice, and then construct an image space based only on the active fibers. Next, all nonzero elements are remapped by re-encoding their coordinates according to the new image space without changing the values of these elements. For example, in Figure 4.3, we first find all active mode fibers (as shown in ❶), identifying that the first time slice has 5, 10, and 4 active fibers (unique index values) along mode-1, mode-2, and mode-3, respectively. After that, an image space of size $5 \times 10 \times 4$ is constructed and all nonzero indices are remapped (as shown in ❷) to the new image space. Although the resulting tensor sketch contains the same 10 nonzero elements, it has more than $46\times$ smaller volume than the original tensor slice, and correspondingly higher density. As a result, the remapped tensor slice is much faster to represent in a compact linearized form (as detailed in 4.3.2) and it also exhibits higher data reuse, which makes it amenable for more efficient parallel execution during STF computations. Once the computation for the current time slice is completed, the image index values are remapped to their original values to update the factor matrices.

While sophisticated tensor permutation [53, 90] can be used to reduce the

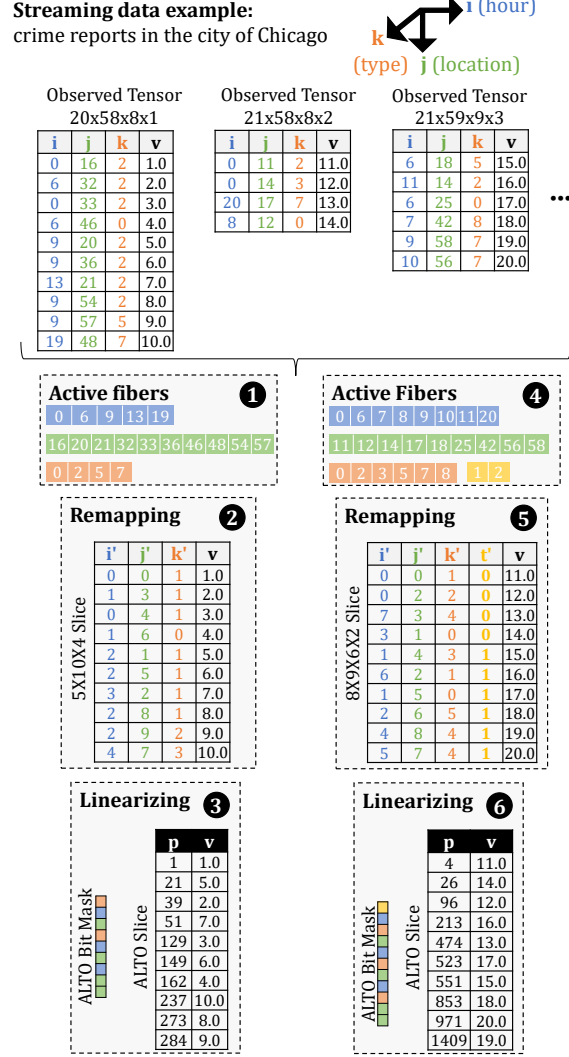


Figure 4.3. Illustration of how spatial sketching, dynamic index linearization, and dynamic time slicing are applied to a set of time slices from the **Chicago** tensor. The incoming time slice is first analyzed to identify active fibers in ①, then remapped to a smaller image space in ②, and the remapped indices are linearized to a compact linear space in ③. If a time slice does not have enough nonzero elements (e.g., second time slice), it is dynamically aggregated with the subsequent time slice(s), and the same sequence of identifying active fibers ④, remapping ⑤, and linearization ⑥ is applied.

irregularity of high-dimensional data, and potentially lead to more efficient spatial sketches, these approaches require expensive multi-dimensional sorting and/or graph/hypergraph-based tensor partitioning, which is not feasible in a streaming setting. Instead, our spatial sketching leverages readily available information about the streaming data. That is, to identify the active fibers (step ❶ in Figure 4.3), our algorithm reuses the output of PARTITION in Algorithm 2. However, finding active fibers (unique indices) can be done independently in linear time (*e.g.*, using hash tables), if it is not already required by the target streaming algorithm.

4.3.2 Dynamic Index Linearization

Prior state-of-the-art STF algorithms use the raw list-based COO format because of the prohibitive cost of constructing compressed tensor representations on-the-fly for every new time slice. The main reason for this expensive format generation is that tree-based (*e.g.*, CSF [90, 87]) and block-based (*e.g.*, HiCOO [52]) formats *require* clustering and partitioning of nonzero elements along one or more modes to achieve compression. Since streaming tensors suffer from even more irregular data distributions and extreme sparsity (see Figure 4.1) than static tensors, the cost of generating such coarse-grained formats outweighs their benefits [32]. However, processing streaming data in a raw form leads to inefficient parallel execution and random memory access. While extensions to the COO format [57, 79] can improve its performance by reducing synchronization cost, the nonzero elements need to be replicated for *every mode* [57] or multiple permutation arrays (one for each mode) need to be stored, which increases memory usage and reduces input data locality [79].

To address the irregular sparsity of time-evolving tensors, we extend the Adap-

tive Linearized Tensor Order (ALTO) format [32] for streaming data. In doing so, we leverage index linearization to compress the data rather than the expensive multi-dimensional clustering and partitioning of nonzero elements, and each nonzero element is accessed using a single index constructed by interleaving its mode bits. While the original ALTO format is much faster to generate than tree- and block-based tensor formats, it still requires several full CP-ALS iterations over the entire tensor to amortize its format generation cost [32, 67], which is not suitable in a streaming setting. As such, we propose several streaming-aware optimizations and extensions to the original ALTO format to allow efficient processing of time-evolving tensors.

First, instead of generating an entirely new linearized tensor for each incoming data batch (time slice), ALTO-stream initially constructs a skeleton tensor (with no data items) along with an index schema and then reuses them over different time instances. This skeleton tensor is generated based on the common (high-level) characteristics across the time slices being analyzed and the target hardware platform (i.e., the type of each tensor mode and the number of tensor modes, data partitions, and compute cores/threads). Specifically, once a batch of nonzero elements is ingested at a given time instance (t), the skeleton tensor at the previous time instance ($t - tt$, where tt is the slice width) is inspected to determine if its reserved memory needs to grow to accommodate the new data, and then the indexing schema is updated according to the data shape (multi-dimensional space) and all incoming nonzero elements are linearized (as detailed in ③ and ⑥ from Figure 4.3). This incremental generation of linearized tensors allows ALTO-stream to substantially reduce the cost of linearizing streaming data on-the-fly. That is, by reusing the skeleton tensor, the format generation cost is largely reduced to the time required to update the indexing schema and to linearize the new nonzero el-

		i'	j'	k'	v		i'	j'	k'	v
thread 0	{	0	0	1	1.0		0	0	1	1.0
		1	3	1	2.0		0	4	1	3.0
thread 1	{	0	4	1	3.0		1	3	1	2.0
		1	6	0	4.0		1	6	0	4.0
thread 2	{	2	1	1	5.0		2	1	1	5.0
		2	5	1	6.0		2	5	1	6.0
thread 3	{	3	2	1	7.0		2	8	1	8.0
		2	8	1	8.0		2	9	2	9.0
thread 4	{	2	9	2	9.0		3	2	1	7.0
		4	7	3	10.0		4	7	3	10.0

Figure 4.4. Illustration of output-oriented traversal of nonzero elements. Table on the right shows the nonzero elements after reordering the left table (from ❷ in Figure 4.3) by mode-1 (output) index values. During parallel update of mode-1 factor matrix, nonzero elements that update the same row are now more likely to be assigned to the same execution thread, reducing synchronization cost.

ements.

Most importantly, by spatially sketching the tensor space (discussed in Sec. 4.3.1) before generating the linearized tensor slice, ALTO-stream dramatically reduces the linear space required to represent a given time slice, maximizing the compression effect of index linearization. For example, as shown in ❸ from Figure 4.3, ALTO-stream uses a linear space of length 512 or a bit mask of 9 bits to encode the linearized indices (p), *i.e.*, 3 bits to encode mode-1 index range (0 – 4), 4 bits to encode mode-2 index range (0 – 9), and 2 bits to encode mode-3 index range (0 – 3). Without spatial sketching, index encoding would require a linear space of length 16,384.

Secondly, ALTO-stream accelerates STF computations (*e.g.*, MTTKRP) by introducing new and alternative tensor traversal and synchronization methods and automatically selecting the most efficient algorithm at runtime based on the target mode type (*i.e.*, streaming/temporal or non-streaming/spatial mode) and the char-

acteristics of tensor slices (*i.e.*, data reuse and spatial dimensions). Specifically, due to the extreme sparsity of streaming data, the STF computations required to update factor matrices often have limited reuse of factor matrix rows (or fibers) and suffer from substantial synchronization cost compared to static tensors. To tackle this issue, ALTO-stream estimates the reuse of factor fibers (factor matrix rows) by computing the average number of nonzero elements per fiber and — if the fiber reuse is low — uses output-oriented tensor traversal to update the target factor matrix.

Figure 4.4 illustrates this output-oriented traversal while updating mode-1 factor matrix, where the left table shows the original ordering of nonzero elements (as shown in ❷ from Figure 4.3) and the right table shows the nonzero elements after reordering by mode-1 (output) index values. In output-oriented traversal, the first two threads (threads 0 and 1) can now process nonzero elements with output (i') index 0 and 1 independently without using expensive atomic operations, as there is no risk of contention with other threads. Atomic operations are required only at the boundaries, *e.g.*, when row 2 is updated, as both thread 2 and thread 3 own nonzero elements with output index 2. While this output-oriented traversal is similar to strategies adopted by other COO-based formats [79, 57], ALTO-stream does not require redundant copies of the tensor or dedicated permutation arrays for every tensor mode. Instead, ALTO-stream only uses output-oriented traversal when the reuse of factor fibers is limited along a certain mode. Otherwise, if a time slice exhibits data reuse of fibers, the original ALTO ordering and traversal (where nonzeros elements are sorted/accessed based on their linearized indices, as shown in ❸ from Figure 4.3) is used along with its adaptive (reduction-based) synchronization strategy, due to their superior data locality and parallel performance [32].

4.3.3 Dynamic Time Slicing

Real-world streaming tensors are subject to highly uneven data distribution across time slices (as shown in Figure 4.1). Thus, the number of updates (nonzero elements) in a time slice can vary widely, ranging from a few updates to millions or hundreds of millions of updates (c.f. Figure 4.7). However, the current state-of-the-art STF algorithms update the factors using fixed-width time slices, regardless of the characteristics of these time slices. Using fixed-width time slices results in poor performance, as time slices with small number of updates lead to under-utilization of computing resources.

To address this problem, we introduce dynamic time slicing (DTS), which aims to reduce the time required for ingesting tensor data and updating factor matrices by increasing the parallelism and data reuse of STF computations. DTS employs variable-width time slices for factorizing streaming high-dimensional data by setting a minimum weight (w nonzero elements) for each time slice used to update the factor matrices based on the target hardware architecture. In addition, DTS allows the user to enforce a target update frequency (T_f time slices), after which the factor matrices must be updated even if the weight of the current time slice is below the minimum weight w . This way, DTS keeps accumulating incoming updates and increasing the width of the current time slice until the minimum number of updates is achieved, or until the user-provided update frequency is reached, whichever occurs first.

In Figure 4.3, the second (*i.e.*, middle) list of updates only has four nonzero elements. For $w = 10$ and $T_f = 2$, it is combined with the third (*i.e.*, right-most) list of updates to create a larger time slice. Once enough nonzero elements have been aggregated, or if the update frequency is reached, the same process of identifying

active fibers (④) and spatially sketching the tensor space (⑤) is applied, followed by index linearization (⑥). However, since the resulting time slice has a variable width and contains nonzero elements or interactions across different points in time, a temporal index (t) is introduced to distinguish between these elements and to accurately compute the temporal factor matrix. Yet, the resulting variable-width slice, represented by a linear space of length 2,048 or a bit mask of length 11 bits, is still much more compact than the original two slices, which are represented by $21 \times 58 \times 8$ and $21 \times 59 \times 9$ spatial spaces.

Even though DTS appears to introduce a temporal mode, variable-width time slices are still treated as a typical time slice in STF. The only difference is that the temporal mode computation (Lines 2 in Algorithm 2) now generates a time *matrix*, where each row corresponds to a respective time instance/slice, instead of a time vector (single row).

This tunable DTS mechanism is particularly useful for production data analysis platforms, where many users share the same system, and making the most efficient use of such expensive computing resources is paramount. In addition, while existing tensor factorization libraries are limited to sparse tensors that fit in memory, ALTO-stream can divide up large-scale tensors and analyze them in a semi-streaming manner even when system memory is limited. If the target user application requires an immediate update of factor matrices or the incoming data cannot be stored for privacy concerns, the user can still use ALTO-stream with the traditional (fixed-width) time slicing approach by setting $T_f = 1$.

4.3.4 Putting It All Together: ALTO-stream

Algorithm 4 shows the high-level algorithm for spCP-stream using the ALTO-stream framework. Note that our optimization strategies are *orthogonal* to spCP-stream, and can be generalized to other STF algorithms. If we eliminate SKETCH and EXPAND (spatial sketching), LINEARIZE (dynamic index linearization), and AGGREGATE (dynamic time slicing), we are left with the original spCP-stream algorithm. ALTO-stream is applied similarly to CP-stream (not shown here for brevity).

4.4 Evaluation

We evaluate the proposed ALTO-stream framework¹ using a diverse set of real-world tensors with temporal (streaming) modes. We compare the achieved performance against the prior state-of-the-art parallel implementation of the CP-stream [86] and spCP-stream [91] algorithms (referred together as SPLATT-stream²). To simulate a streaming setting, we read tensor data from disk, and then process the data into temporal tensor slices in a raw (COO) form and feed these slices into the STF frameworks (ALTO-stream and SPLATT-stream). We report the performance by measuring the time required to ingest the temporal slices and to generate/update the factor matrices. For a fair comparison, this time includes the on-the-fly format generation cost for ALTO-stream.

¹Available at <https://github.com/jeewhanchoi/ALTO-stream>

²Available at: <https://github.com/jeewhanchoi/row-sparse-cpstream>

Algorithm 4 High-level algorithm for ALTO-stream with *spatial sketching*, *dynamic index linearization*, and *dynamic time slicing* applied to spCP-stream. After AGGREGATING time slices based on some minimum weight (w) and target update frequency (T_f), a new tensor time slice ($\hat{\mathcal{X}}$) is generated, along with information on the slice width (tt). Then, the resulting time slice and factor matrices are spatially SKETCHED, and the nonzero elements are LINEARIZED based on the sketched space. Once the current time slice has been entirely processed, the sketched factor matrices are EXPANDED and merged to generate the updated factor matrices.

Input: (1) $(N - 1)$ -mode tensor stream $\mathcal{X}_t$ for $t = 1, \dots, T$, (2) minimum slice weight w , and (3) target update frequency T_f

Output: Factor matrices $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N-1)}$ and $\mathbf{S}$ (time-mode)

```

1: for  $t = 1, \dots, T$  do
2:    $[done, \hat{\mathcal{X}}, tt] \leftarrow \text{AGGREGATE}(\mathcal{X}_t, w, T_f)$ 
3:   if not  $done$  then
4:     continue ▷ Aggregate until  $w$  or  $T_f$  is reached
5:   end if
6:   for  $n = 1, \dots, N - 1$  do
7:      $[\mathbf{A}_{\alpha,t}^{(n)}, \mathbf{A}_{\beta,t}^{(n)}] \leftarrow \text{PARTITION}(\hat{\mathcal{X}})$ 
8:      $[\hat{\mathcal{X}}', \mathbf{P}_{\alpha,t}^{(n)}] \leftarrow \text{SKETCH}(\mathbf{A}_{\alpha,t}^{(n)}, \hat{\mathcal{X}})$ 
9:      $\hat{\mathcal{X}}' \leftarrow \text{LINEARIZE}(\hat{\mathcal{X}}', \mathbf{P}_{\alpha,t}^{(n)})$ 
10:  end for
11:  repeat
12:     $\mathbf{S}_t \leftarrow \text{INSTANT\_UPDATE}(\mathbf{P}_{\alpha,t}^{(v)}, \hat{\mathcal{X}}')$ 
13:    for  $n = 1, \dots, N - 1$  do
14:       $\mathbf{P}_{\alpha,t}^{(n)} \leftarrow \text{UPDATE}(\mathbf{P}_{\alpha,t}^{(v)}, \mathbf{P}_{\alpha,t-tt}^{(n)}, \hat{\mathcal{X}}')$ 
15:       $\mathbf{A}_{\beta,t}^{(n)} \leftarrow \text{UPDATE}(\mathbf{A}_{\beta,t}^{(v)}, \mathbf{A}_{\beta,t-tt}^{(n)})$ 
16:    end for
17:  until convergence or max. # of iterations reached
18:   $\mathbf{A}_{\alpha,t}^{(n)} \leftarrow \text{EXPAND}(\mathbf{P}_{\alpha,t}^{(n)}) \forall n \in \{1, \dots, N - 1\}$ 
19:   $\mathbf{A}_t^{(n)} \leftarrow \text{MERGE}(\mathbf{A}_{\alpha,t}^{(n)}, \mathbf{A}_{\beta,t}^{(n)}) \forall n \in \{1, \dots, N - 1\}$ 
20:   $\mathbf{S} \leftarrow \text{CONCATENATE}(\mathbf{S}_{t-tt}, \mathbf{S}_t)$ 
21: end for
22: return  $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N-1)}, \mathbf{S}$ 

```

Table 4.1. The target streaming sparse tensors, ordered by the number of nonzero elements. The temporal mode is underlined.

Tensor	Dimensions	NNZ	Density
NIPS	$2.5K \times 2.9K \times 14K \times \underline{17}$	$3.1M$	$1.8e-06$
Uber	$\underline{183} \times 24 \times 1.1K \times 1.7K$	$3.3M$	$3.8e-04$
Chicago	$\underline{6.2K} \times 24 \times 77 \times 32$	$5.3M$	$1.5e-02$
LANL	$\underline{3.8K} \times 11.2K \times 8.7K \times 75.1K \times 9$	$69.1M$	$2.8e-10$
Flickr	$319.7K \times 28.2M \times 1.6M \times \underline{731}$	$112.9M$	$1.1e-14$
Delicious	$532.9K \times 17.3M \times 2.5M \times \underline{1.4K}$	$140.1M$	$4.3e-15$
Patents	$\underline{46} \times 239.2K \times 239.2K$	$3.6B$	$1.4e-03$

4.4.1 Experimental Setup

4.4.1.1 Platform

The evaluation platform is an Intel Xeon Platinum 8280 CPU, based on the Cascade Lake micro architecture. It has two sockets, each with 28 cores, and the parallel experiments use all the available 56 cores. For accurate performance measurements, the target CPU runs at a fixed base frequency of 1.8GHz without turbo mode. The server runs CentOS 7.6 Linux distribution, and it has 384 GiB of main memory. Both STF frameworks are compiled using Intel C/C++ compiler (version 19.1.3) with -O3 optimization level.

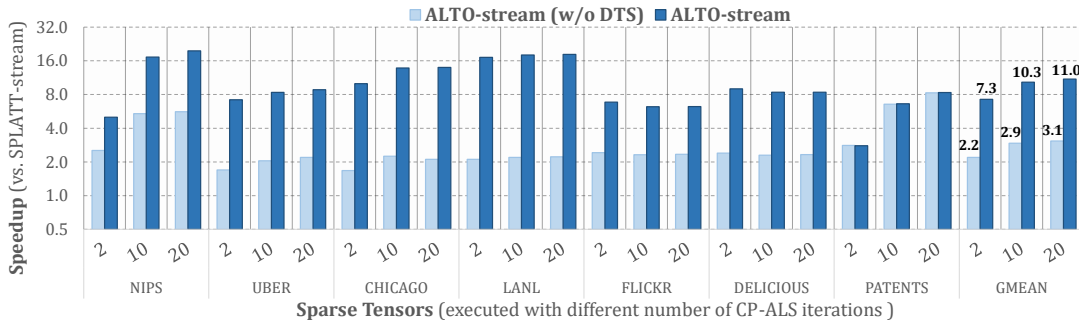


Figure 4.5. The performance of CP-stream algorithm using ALTO-stream compared to the state-of-the-art SPLATT-stream on an Intel Xeon Platinum 8280 (“Cascade Lake”) CPU with 56 cores.

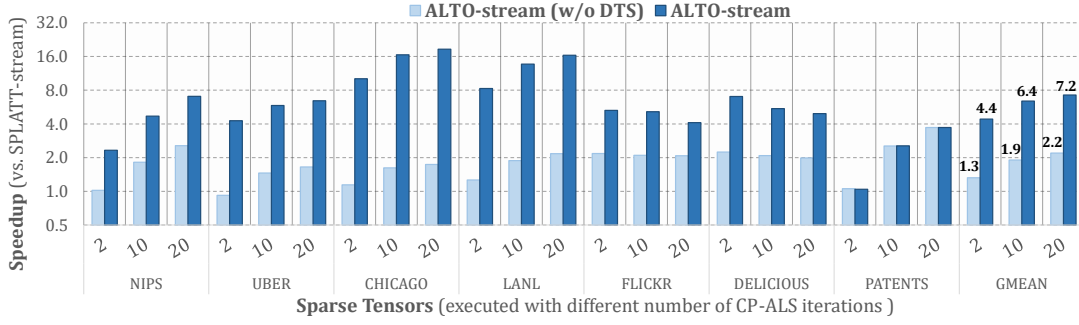


Figure 4.6. The performance of spCP-stream algorithm using ALTO-stream compared to the state-of-the-art SPLATT-stream on an Intel Xeon Platinum 8280 (“Cascade Lake”) CPU with 56 cores.

4.4.1.2 Datasets

The experiments use *seven* real-world sparse tensors with streaming (temporal) modes, taken from the FROSTT [85] and Advanced Research in Cyber Systems [99] open-source repositories (see Table 4.1). They cover a wide range of data properties—number of modes, mode lengths, number of nonzero elements, and density. Table 4.1 shows a summary of these sparse tensors, sorted by the number of nonzero elements. The temporal mode for each tensor is underlined.

4.4.1.3 Configurations

The experiments use double-precision arithmetic and 64-bit integers. We configure ALTO-stream to automatically select the appropriate length of its linearized index (64 or 128 bits) based on the characteristics of each time slice. Although the proposed DTS strategy can benefit from auto-tuning, we set the minimum slice weight (w) to 2^{20} nonzero elements and the update frequency (T_f) to 16 time slices for simplicity. Since ALTO-stream constructs its compact tensor representation on the fly (*i.e.*, while ingesting incoming tensor data), the number of iterations required to update the factors can affect the performance. However, the number of iterations can vary depending on various factors (*e.g.*, data set, factorization rank,

convergence tolerance, and randomly initialized factors). Prior studies showed that CP-ALS uses around 16 iterations on average for real-world sparse tensors [95]. Our empirical study indicates that CP-ALS takes on average 7 to 65 iterations to compute the factors of the target time-evolving tensors in a streaming setting. For a fair comparison against SPLATT-stream, each dataset is executed for a specific number of iterations regardless of the initial factor matrices. We evaluate the performance across a conservative range of CP-ALS iterations, namely, between 2 and 20 iterations, where 2 is the minimum number of iterations required for convergence. For accuracy validation, we use a convergence tolerance of 10^{-5} and run each data set with five different randomly initialized factors. While each data set typically requires an application-specific factorization rank (R), we set R to 16 for a fair evaluation across data sets.

4.4.2 Results

To illustrate the performance of ALTO-stream, Figures 4.5 and 4.6 show the speedup achieved compared to SPLATT-stream with and without DTS, respectively. That is, “ALTO-stream (w/o DTS)” demonstrates the performance of our dynamic tensor linearization, while “ALTO-stream” shows the performance of both dynamic tensor linearization and dynamic time slicing. For the typical number of CP-ALS iterations (10–20), ALTO-stream delivers substantial performance gains compared to the state-of-the-art approach, achieving $10.3\text{--}11\times$ and $6.4\text{--}7.2\times$ geometric-mean speedup for the CP-stream and spCP-stream algorithms, respectively. Compared to CP-stream, spCP-stream reduces the number of compute/memory operations required to update the factor matrices via a sparsity-aware partitioning of these factors. Hence, the format generation overhead of ALTO-stream has a higher

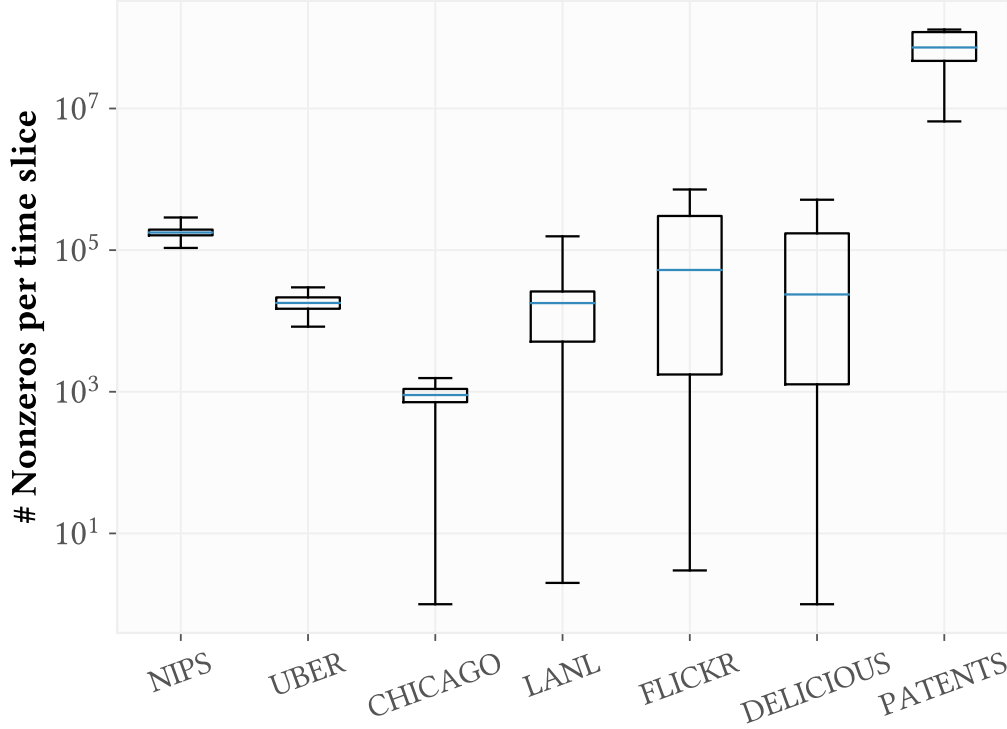


Figure 4.7. A box plot showing the varying number of nonzero elements across temporal tensor slices.

performance impact, which decreases the overall speedup relative to the CP-stream algorithm. Due to the cost of dynamically generating linearized tensor representations, the performance of ALTO-stream improves as the number of iterations increases, except for extremely sparse tensors (e.g., *Delicious*) as the format generation overhead is relatively low compared to the time required to compute/update the factor matrices. However, even for the minimum number of iterations required for convergence (*i.e.*, 2), ALTO-stream still realizes $7.3\times$ and $4.4\times$ geometric-mean speedup for CP-stream and spCP-stream, respectively.

Further analysis indicates that our DTS strategy increases data reuse and parallelism during STF computations, which leads to more efficient utilization of computing resources. As a result, it delivers $3.3\text{--}3.6\times$ *additional* performance improvement on top of the dynamic tensor linearization method. Specifically, DTS sub-

stantially increases resource utilization when the majority of time slices have a limited number of interactions (nonzero elements), as detailed in Figure 4.7. For the **Patents** data set, DTS has negligible performance impact as the average number of interactions per slice is large.

4.4.2.1 Varying Factorization Rank

Factorization rank (i.e., number of latent features or tensor components) affects the number of operations and iterations required to compute the CP-ALS model of a given tensor. Moreover, higher factorization ranks increase the spatial data locality of STF computations, due to wider factor rows, and lead to more useful work per incoming nonzero element (or interaction). Although a tensor is unlikely to have one component, using a factorization rank of 1 will turn all tensor factors into vectors rather than matrices. To better understand the performance impact of the factorization rank, we characterized the performance of spCP-stream algorithm by varying the factorization rank from 1 to 64. Figure 4.8 demonstrates the performance of ALTO-stream (without DTS) compared to SPLATT-stream across different factorization ranks for a set of representative tensors. Since the factor matrices of **Chicago** are extremely small, varying the factorization rank has a negligible effect on the data reuse (*i.e.*, data is found in the cache at all rank sizes). Hence, doing more useful work per nonzero element (higher ranks and more CP-ALS iterations) improves the performance of ALTO-stream due to better amortization of the dynamic linearization overhead. In contrast, the **LANL** dataset has large factor matrices, and it is highly sparse. As a result, the dynamic linearization overhead of ALTO-stream has limited performance impact and decreasing the factorization rank leads to higher speedup compared to SPLATT-stream, due to the better data reuse and more effective cache and memory utilization of ALTO-

stream. While the **Patents** data set has large factor matrices similar to **LANL**, it has higher density and large number of nonzero elements per slice. Therefore, it shows a similar performance trend to **LANL** except when the amount of work per nonzero element is low; that is, the relatively high dynamic linearization cost in the **Patents** data set leads to lower performance when using a low factorization rank (rank-1) and a small number of CP-ALS iterations (2).



Figure 4.8. The performance of spCP-stream algorithm using ALTO-stream (without DTS) compared to the state-of-the-art SPLATT-stream across different factorization ranks (or tensor components).

4.4.2.2 Memory Traffic Analysis

Since the performance of STF is primarily determined by the memory traffic and throughput [90, 91], we collected and characterized the memory-access metrics using the ITT (Instrumentation and Tracing Technology) APIs in the Intel VTune Profiler [35]. Table 4.2 details the memory-related metrics of the spCP-stream algorithm across ALTO-stream and SPLATT-stream for two representative data sets. The analysis shows that ALTO-stream improves memory performance by (i) processing the data in a compact representation rather than a raw form, (ii) improving data locality and cache utilization during STF computations, and (iii) reusing factorization data across time slices. As a result, ALTO-stream substantially re-

Table 4.2. The memory metrics (loads/stores) of spCP-stream across the STF frameworks. The number of CP-ALS iterations is 2.

Tensor	Framework	LD/ST Count	LD/ST Rate
Chicago	ALTO-stream	5.22e+10	3.42e+10 /s
	ALTO-stream w/o DTS	48.28e+10	3.57e+10 /s
	SPLATT-stream	50.51e+10	3.27e+10 /s
LANL	ALTO-stream	21.34e+10	3.84e+10 /s
	ALTO-stream w/o DTS	77.98e+10	2.14e+10 /s
	SPLATT-stream	93.17e+10	2.02e+10 /s

duces the memory traffic (LD/ST count) and increases the memory throughput (LD/ST rate) compared to SPLATT-stream. For the **Chicago** data set, the factor matrices fit in cache and the majority of tensor slices are small (c.f. Figure 4.7). As such, the compact data representation and reuse of factorization data across slices decrease the memory traffic, but the memory throughput (LD/ST rate) is comparable to SPLATT-stream. Furthermore, the memory metrics show that dynamic time slicing yields a significant reduction in memory traffic when time slices have a limited number of nonzero elements (e.g., **Chicago**) and considerable improvement in memory throughput for highly sparse tensors with large factor matrices (e.g., **LANL**).

4.4.3 Model Validation

To evaluate the factorization quality of the STF frameworks, we use the global relative fitting error, as per the original CP-stream study [86], which is analogous to the mean squared error in principal component analysis (PCA):

$$\frac{1}{\|\mathcal{X}\|^2} \left\| \mathcal{X} - \llbracket \{\mathbf{A}^{(n)}\}; \mathbf{S} \rrbracket \right\|^2 \quad (4.1)$$

The global relative fitting error is computed by reconstructing the tensor from the final factor matrices (*i.e.*, factor matrices computed at time T) and the temporal factor matrix (*i.e.*, $\mathbf{S}$), and comparing it to the complete static (spatio-temporal)

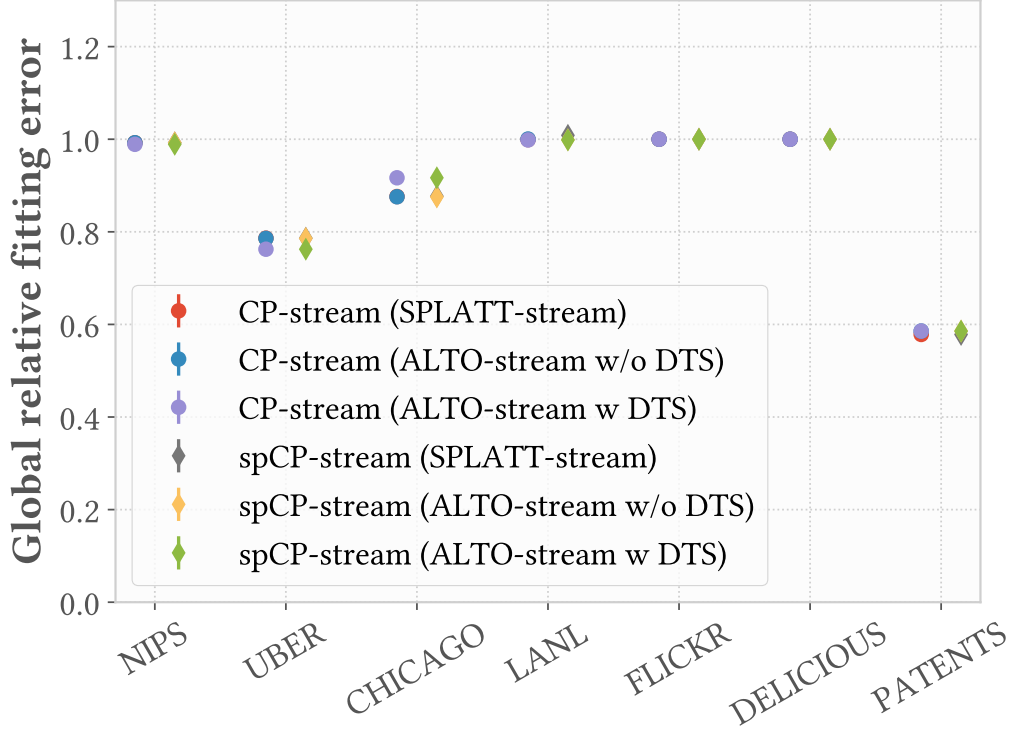


Figure 4.9. Comparison of the global relative fitting error between SPLATT-stream and ALTO-stream, with and without DTS. ALTO-stream demonstrates comparable accuracy to SPLATT-stream across the streaming tensor data sets.

tensor. Since the factorization quality highly depends on the random initial factors, as well as the factorization rank and convergence tolerance, we report the median error across five randomly initialized factors. Figure 4.9 demonstrates that ALTO-stream preserves the accuracy of STF algorithms and it has comparable global relative fitting error to SPLATT-stream.

4.5 Conclusion

In this chapter, we address our second research question: **RQ2**: Can we further improve the throughput of streaming sparse tensor factorization utilizing adaptive linearized sparse tensor formats? We make the following contributions:

1. We present a *dynamic tensor linearization* scheme that quickly generates com-

pressed tensor slices on-the-fly and selects the best nonzero element traversal and synchronization strategy based on the characteristics of each slice. This is accomplished by using a *spatial sketching* algorithm that retains all incoming nonzero elements at a given time instance but remaps them into a tensor sketch with reduced multi-dimensional image space. The coordinates are then linearized into a single index and the nonzero elements are sorted accordingly. This compact format greatly reduces memory usage and improves the data locality of STF with minimal format generation overhead. (Sec. 4.3.1 and 4.3.2).

2. We introduce dynamic time slicing (DTS), a tunable mechanism to balance the frequency of factor updates and the efficient utilization of computing resources. DTS uses variable-width time slices (instead of a fixed width) by aggregating interactions across time slices until some minimum slice weight (*i.e.*, number of nonzero elements) is achieved or a user-provided update frequency is reached. This improves the overall resource utilization, and in turn increases the achieved performance, while maintaining the accuracy of the original algorithm. (Sec. 4.3.3).
3. We demonstrate the efficacy of our STF framework by accelerating the state-of-the-art CP-stream [86] and spCP-stream algorithms [91], and improving their performance by $10.3 - 11\times$ and $6.4 - 7.2\times$, respectively, for a range of real-world streaming tensors on a 56-core CPU. (Sec. 4.4.2).

CHAPTER 5

FULLY GPU-ACCELERATED CONSTRAINED SPARSE TENSOR FACTORIZATION

This chapter includes material previously published at the 2024 International Conference of Parallel Processing (ICPP) in Section 5.2, Section 5.3, Section 5.4. This work was done as a collaborative effort involving the University of Oregon and Oak Ridge National Laboratory. As the first author, I was responsible for the implementation, experimentation and the drafting of the manuscript. Co-authors Dr. Ramakrishnan Kannan, Dr. Piyush Sao, and Dr. Jee Choi provided technical guidance and assisted with the reviewing and editing of the manuscript.

5.1 Problem Statement

In this section, we address our third research question: **RQ3**: With the availability of modern, powerful GPUs, can we utilize these parallel computing architectures by implementing a fully GPU-resident constrained sparse tensor factorization framework?

5.2 Motivation

The handling of highly sparse nature of real-world tensors, which leads to irregular memory access, workload imbalance, and synchronization overhead make computation *involving sparse tensors* particularly challenging to execute efficiently on massively parallel GPU architectures. This is demonstrated by the matricized tensor times Khatri-Rao product (MTTKRP) operation, which is the primary performance bottlenecks in most *unconstrained* STF algorithms for both CPUs [90, 32] and GPUs [67, 69].

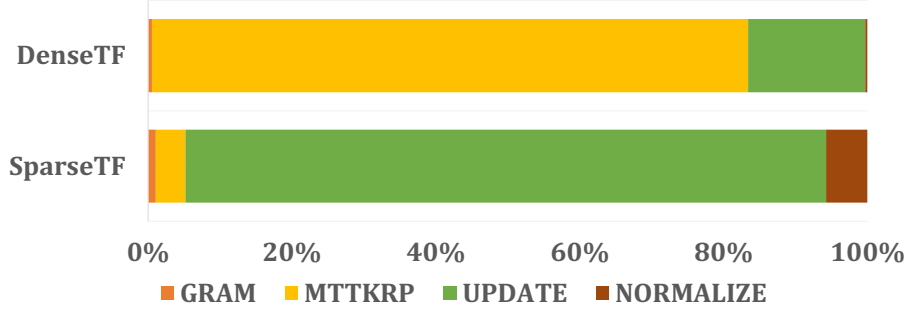


Figure 5.1. Execution time breakdown of constrained tensor factorization for dense (**DenseTF**) and sparse (**SparseTF**) tensors using the ADMM update methods from the PLANC library [23]. For dense tensors, MTTKRP dominates the overall execution time. However, for sparse tensors, the ADMM update process for Non-negative CP factorization takes significantly longer than MTTKRP. The dense tensor is a synthetic $400 \times 200 \times 100 \times 50$ tensor, while the sparse tensor is the *Delicious* tensor (Table 2.1). A factorization rank of $R = 32$ is used for both.

An additional challenge comes from accomodating the *constraints* inherent in many real-world tensors. For instance, multi-dimensional sensory input data often comprises of *non-negative* values. Therefore, imposing such constraints on the factors typically results in a more interpretable output for domain scientists, making it a crucial aspect of tensor analysis. Various update methods are used on the *dense* factor matrices to impose constraints, and several optimized libraries exist for CPUs [23, 84]. However, as far as we are aware, no studies have been conducted on optimizing update methods for constrained sparse tensor factorization (cSTF) on massively parallel GPU architectures.

As such, we draw on prior studies on CPUs for constrained tensor factorization to estimate the performance characteristics of cSTF and the impact of the update methods on the overall performance on GPUs. Figure 5.1 (under **DenseTF**) shows a breakdown of execution time for *constrained dense* tensor factorization using three different update methods on a synthetic $400 \times 200 \times 100 \times 50$ tensor with a factorization rank of $R = 32$, obtained using the PLANC library [23]. For dense

tensors, the MTTKRP operation dominates the overall execution time. We can attribute this to the proportionally larger size of the dense tensor ($400 \cdot 200 \cdot 100 \cdot 50$ elements) compared to its factor matrices ($(400 + 200 + 100 + 50) \cdot R$ elements in total where R is the factorization rank).

In contrast, real-world sparse tensors demonstrate very high levels of sparsity, where the size of the tensors (i.e., number of non-zero elements) is comparable to the size of the factor matrices (see Table 2.1 for examples of sparse tensors). Therefore, we expect the update process to be at least as expensive as the MTTKRP operation. To test our hypothesis, we modified the PLANC library to execute constrained sparse tensor factorization on the `Delicious` sparse tensor using the same update method and factorization rank ($R = 32$). Figure 5.1 (under **SparseTF**) shows a breakdown of execution time. As expected, we see that the update process is *significantly more expensive* than the MTTKRP operation.

From this preliminary study, we hypothesize that offloading the entire cSTF algorithm to the GPU will significantly improve performance compared to CPU-based implementations. The update process operates over the *dense* factor matrices with regular memory access, taking full advantage of the higher memory bandwidth and compute performance of modern GPUs. Offloading the entire end-to-end cSTF computation to the GPU eliminates the need to transfer data between host and GPU over the slower PCIe or NVLink interconnect. While the MTTKRP operation on the sparse tensor is also a significant performance bottleneck for cSTF, existing GPU work [67, 69] can be leveraged to further reduce the end-to-end execution time.

Algorithm 5 GPU Optimized cuADMM Algorithm with *Preinversion* and *Operation fusion*. With *Preinversion*, an explicit inverse can be calculated using a Cholesky solve outside the inner loop (line 4), and the multiple triangular solve can be replaced with a DGEMM operation (line 7).

Input: $\mathbf{M}, \mathbf{S}, \mathbf{H}, \mathbf{U}, R$ $\triangleright \mathbf{M}$ is the MTTKRP matrix, R is the rank of the factors, $\mathbf{S}$ is the Hadamard of gram matrix, $\mathbf{H}, \mathbf{U}$ are the primal and dual variables respectively

- 1: Initialize $\mathbf{H}$ and $\mathbf{U}$
- 2: $\rho \leftarrow \text{trace}(\mathbf{S})/R$
- 3: Calculate $\mathbf{L}$ from the Cholesky decomp. of $\mathbf{S} + \rho\mathbf{I} = \mathbf{L}\mathbf{L}^T$
- 4: $(\mathbf{L}\mathbf{L}^T)^{-1} \leftarrow$ Explicit inverse using Cholesky solve
- 5: **repeat**
- 6: $\tilde{\mathbf{H}} \leftarrow \text{compute_auxiliary}(\mathbf{M}, \mathbf{H}, \mathbf{U}, \rho)$
- 7: $\tilde{\tilde{\mathbf{H}}} \leftarrow (\mathbf{L}\mathbf{L}^T)^{-1}\tilde{\mathbf{H}}$
- 8: $\mathbf{H} \leftarrow \text{apply_proximity_operator}(\mathbf{H}, \tilde{\mathbf{H}}, \mathbf{U})$
- 9: $\mathbf{U}, \Delta\mathbf{H} \leftarrow \text{dual_update}(\mathbf{U}, \mathbf{H}, \tilde{\mathbf{H}})$
- 10: **until** $\|\Delta\mathbf{H}\|_F^2/\|\mathbf{H}\|_F^2 < \epsilon$ **and** $\|\mathbf{H} - \mathbf{H}_0\|_F^2/\|\mathbf{U}\|_F^2 < \epsilon$
- 11: **return** $\mathbf{H}$

5.3 Methodology: Fully GPU-Accelerated Constrained Sparse Tensor Factorization

We extended the CPU-based PLANC library [23], which supports various constrained update methods for *dense* tensor factorization, to handle sparse tensors and their factorization. This was achieved by incorporating the state-of-the-art MTTKRP algorithm for CPUs [32] that utilizes the adaptive linearized tensor order (ALTO) sparse tensor format. This adaptation forms the basis for our initial analysis in Section 5.2 and lays the groundwork for subsequent GPU optimization. Our GPU cSTF framework leverages the blocked linearized coordinate (BLCO) sparse tensor format [67] for the MTTKRP operation, which is the state-of-the-art on GPUs. The remaining three operations—computing the Gram matrix, ADMM update, and normalization of the factor matrices—are implemented using our own CUDA implementations and cuBLAS kernels.

To accommodate various alternating update schemes, we developed the *Alternating Update Non-negative Tensor Factorization (AUNTF_GPU)* class, which is designed to operate exclusively on the GPU, handling factor matrices and matricized tensor times Khatri-Rao product (MTTKRP) results, and can be further extended to support a diverse set of updates schemes, in addition to ADMM. We will refer to our own framework as cSTF-GPU in the remainder of this paper.

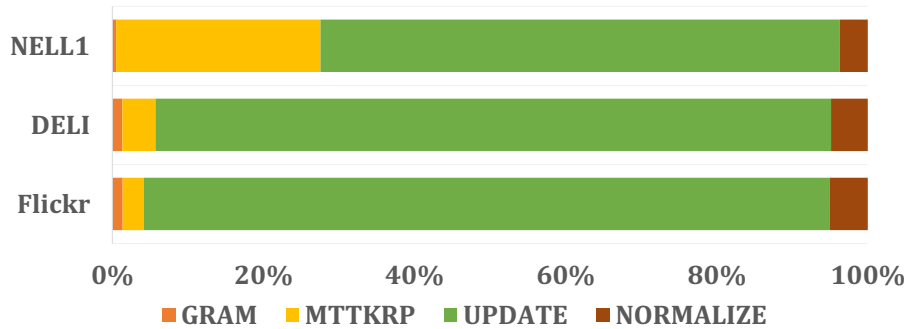


Figure 5.2. Execution time breakdown of cSTF on three tensors with the largest number of non-zero elements—Flickr, Delicious (denoted DELI), and NELL1. Their execution time breakdown is representative of the tensors used in our evaluation, and illustrates that ADMM update dominates the overall execution time.

5.3.1 Profiling the baseline cSTF implementation

We used our modified PLANC library to analyze the performance of cSTF on 10 real-world sparse tensors (Table 2.1). Our analysis in Figure 5.1 shows a significant change in the performance bottleneck when dealing with sparse tensors, with the ADMM update phase becoming the main bottleneck instead of the MTTKRP phase, which is the case for dense tensor factorization. Figure 5.2 illustrates the execution time breakdown for three largest sparse tensors—Flickr, Delicious, and NELL1. This execution time breakdown is representative of the tensors used in our evaluation, providing further evidence that ADMM update dominates the overall

execution time for cSTF, and highlighting the importance of optimizing the update algorithm for cSTF.

5.3.2 cuADMM - GPU-optimized ADMM algorithm

While we address the specific details of the ADMM algorithm in Section 6.2.2, for the purposes of this section, we proceed with the assumption that every operation in the ADMM algorithm can be implemented using highly optimized cuBLAS kernels (e.g., DGEMM, DGEMV, and DGEAM) on GPUs [43]. Under this assumption, it is reasonable to expect that the baseline ADMM algorithm will deliver close to peak performance on GPUs.

However, when these operations compute over large factor matrices, significant memory traffic can occur while reading and writing the *intermediate data* between kernel calls. This fact can easily be overlooked, leading to less-than-efficient implementations.

In shared memory systems, attempts have been made to optimize ADMM updates in the tensor factorization domain through blockwise reformulation, enhancing the temporal locality of matrices [84, 91]. However, these techniques are not effective on GPU architectures as GPU architectures are optimized for large, uniform memory access patterns and parallel workloads, rather than the segmented, blockwise computations.

An additional consideration for optimizing performance on GPU architectures is addressing the degradation caused by Cholesky or triangular solves. This computation type involves solving systems of equations with triangular matrices (either lower or upper) through sequential forward or backward substitutions. The inherently serialized nature of triangular solves conflicts with GPU capabilities, which

excel in parallel processing across multiple data elements (i.e., SIMD or vectorized computation). This mismatch can lead to significant GPU core idle times and sub-optimal performance.

In the following subsection, we outline two performance optimization strategies to address these issues. The impact of these two optimization strategies are evaluated in Section 5.4.1.

5.3.3 Optimized ADMM updates Implementation

Algorithm 5 outlines the optimized ADMM algorithm with our two key optimizations: operation fusion and pre-inversion. In this section, we focus on the implementation details of these optimizations. Since Chapter 6 is dedicated to scalable constrained solvers (NNLS), we defer a comprehensive description of the ADMM algorithm itself to Section 6.2.2.

5.3.3.1 Operation Fusion

To mitigate the overhead of reading and writing intermediate data, we leverage *operation fusion*. Operation fusion is a technique that combines multiple operations into a single kernel, which minimizes both the overhead associated with excessive kernel invocation and access to global memory for intermediate data. Instead of writing the intermediate data to the slow global memory, it is stored in registers and/or shared memory, which can be accessed much more quickly. This also reduces the frequency of global memory accesses, alleviating the stress on the memory system. This is particularly advantageous when processing large factor matrices, as it significantly improves computational throughput.

To address these challenges effectively, we have restructured the computational

graph by developing custom GPU kernels. Specifically, our `compute_auxiliary` kernel integrates the calculation of the auxiliary variable $\tilde{\mathbf{H}} = \mathbf{M} + \rho(\mathbf{H} + \mathbf{U})$, as specified in Line 6 of Algorithm 5. Typically, using a DGEAM kernel for matrix addition, the operation $\mathbf{H} + \mathbf{U}$ would require $2IR$ reads and IR writes. Adding $\mathbf{M}$ would necessitate another DGEAM call, leading to an additional $2IR$ reads and an extra IR write. By fusing these operations into a single kernel, the memory transactions can be reduced to $3IR$ reads and IR writes, effectively decreasing the memory operations by approximately 33%.

In Line 8, the `apply_proximity_operator` kernel concurrently applies the proximity operator to $\tilde{\mathbf{H}} - \mathbf{U}$ and updates the primal variable $\mathbf{H}$. This kernel optimizes memory usage by eliminating the need for intermediate storage of $\tilde{\mathbf{H}} - \mathbf{U}$. Instead, it directly applies the proximity operator and writes the results to $\mathbf{H}$. This implementation leverages the element-wise nature of many proximity operators for various constraints. Specifically, for the non-negativity constraint, the proximity operator functions as an indicator function over $\mathbb{R}_+$, effectively mapping all negative values to zero.

The `dual_update` kernel, mentioned in Line 9, capitalizes on the shared dependencies between the dual variable update $\mathbf{U} = \mathbf{U} + \mathbf{H} - \tilde{\mathbf{H}}$ and the convergence condition computation $\|\mathbf{H} - \tilde{\mathbf{H}}\|$. By reutilizing the intermediate computation of $\mathbf{H} - \tilde{\mathbf{H}}$, this approach conserves both computational resources and memory reads by reducing them by IR each.

5.3.3.2 Pre-inversion

In iterative update frameworks such as ADMM, we often need to solve large linear systems denoted by $\mathbf{A}\mathbf{X} = \mathbf{B}$ during each iteration. This process involves using Cholesky decomposition to extract the $\mathbf{L}$ and $\mathbf{L}^T$ factors of the matrix $\mathbf{S} + \rho\mathbf{I}$

Table 5.1. Hardware and software setup for the evaluation in Chapter 5

	CPU	GPU	
Model	Intel Xeon Platinum 8367HC	NVIDIA A100	NVIDIA H100
u-arch	Ice Lake (ICX)	Ampere	Hopper
Frequency	3.2 GHz	1.41 GHz	1.98 GHz
Cores	26	108 (SM) 6912 (CC)	114 (SM) 14592 (CC)
Caches	3.3MB L1D, 104MB L2, 143MB L3	20.3MB L1D, 40MB L2	28.5MB L1D 50MB L2
DRAM (Bandwidth)	400 GB	80 GB (2039 GB/s)	80 GB (2039 GB/s)
OS/Driver	Ubuntu 20.04	525.85.12	535.54.03
Compiler	gcc 9.3.0	nvcc 11.7	nvcc 12.3

(line 3 in Algorithm 6). We then perform backward and forward substitutions (line 6) to apply the inverse of $\mathbf{S} + \rho\mathbf{I}$. However, the dense triangular solve process is inherently sequential, leading to low computational throughput.

To address this issue, we pre-compute the inverse (line 4) of the matrix, $(\mathbf{L}\mathbf{L}^T)^{-1}$. This allows us to replace multiple triangular solves with a single general matrix-multiply operation (line 7). We only need to compute $(\mathbf{L}\mathbf{L}^T)^{-1}$ once and reuse it in all subsequent steps. This approach maintains the same computational complexity as Cholesky solve but leverages the high efficiency of DGEMM operations on modern GPU architectures.

When using this strategy, we must consider the potential for numerical instability when computing an explicit inverse, especially for close-to-singular matrices. However, in the ADMM framework, the formulation of the matrix $\mathbf{S} + \rho\mathbf{I}$ —comprising the Hadamard products of $\mathbf{H}^T\mathbf{H}$, where $\mathbf{H}$ is a tall and skinny matrix, combined with the stabilizing effect of diagonal loading via adding $\rho\mathbf{I}$ —is empirically observed to be well-conditioned.

5.4 Evaluation

The hardware specifications for our experiments are detailed in Table 5.1. Our study utilizes a range of real-world sparse tensor datasets of varying sizes and sparsity levels, sourced from the FROSTT repository [85], as shown in Table 2.1. We conduct tensor factorization using ranks $\{16, 32, 64\}$ and take the average of 10 execution times for each configuration. We also fix the number of ADMM iterations to 10 for a fair performance comparison across the different frameworks, and since ADMM converges in approximately 10 iterations for all practical purposes.

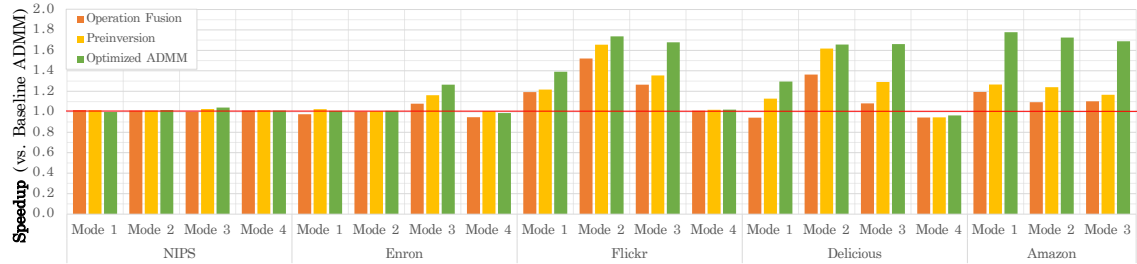


Figure 5.3. Speedup achieved by cuADMM over the baseline ADMM implementation that uses cuBLAS and cuSOLVER kernels. The orange, yellow, and green bars correspond to the speedup achieved by operation fusion (OF), pre-inversion (PI), and both OF and PI, respectively. The datasets are categorized into three groups based on factor matrix size: small (NIPS), medium (Enron), and large (Flickr, Delicious, and Amazon). Speedup is correlated to the factor matrix size, and can be as high as $1.8\times$.

5.4.1 Optimized ADMM performance analysis

We evaluate the impact of our two optimization strategies—*operation fusion* (OF) (Section 5.3.3.1) and *pre-inversion* (PI) (Section 5.3.3.2)—by comparing their performance against the basic ADMM GPU implementation. The basic ADMM implementation uses the highly optimized cuBLAS kernels for its computation. Figure 5.3 displays the speedup achieved when our optimization strategies are applied separately (shown as orange and yellow bars for OF and PI, respectively) and applied

together (shown as green bars) for a single ADMM iteration across a set of representative real-world sparse tensors for a rank-32 factorization.

In general, pre-inversion has a higher impact on performance than operation fusion, and combining both optimization always yields better performance than applying just one or the other. When both OF and PI are applied, little to no speedup (1.0–1.3 $\times$) is achieved for tensors with small (**NIPS**) and medium (**Enron**) factor matrices (i.e., mode lengths), but substantial speedup (up to 1.8 $\times$) is achieved for tensors with large (**Flickr**, **Delicious**, and **Amazon**) factor matrices.

In the rest of this section, our framework will include these two performance optimization strategies when comparing to other frameworks.

5.4.2 Comparison against the state-of-the-art on the latest NVIDIA GPUs

The current state-of-the-art in high-performance cSTF that utilizes ADMM is the *CPU-only* SPLATT library [89]. There are very few studies in cSTF and ADMM on GPUs, and they are not focused on performance and/or the code is not available to the public. As such, we believe SPLATT represents the best comparison for this study, and comparing against SPLATT helps illustrate the benefit of GPU acceleration for cSTF.

In a single iteration of cSTF, four principal computations are required: *Gram*, *MTTKRP*, *ADMM update*, and *normalization*. In this section, we concentrate on the overall end-to-end speedup for a single iteration, with particular emphasis on the improvements in *MTTKRP* and *ADMM update* phases, which are the two primary performance bottlenecks. While optimizing MTTKRP on the GPU is not part of our research contributions, it is important to illustrate its performance against

that of ADMM, as the two algorithms demonstrate a trade-off relationship in performance.

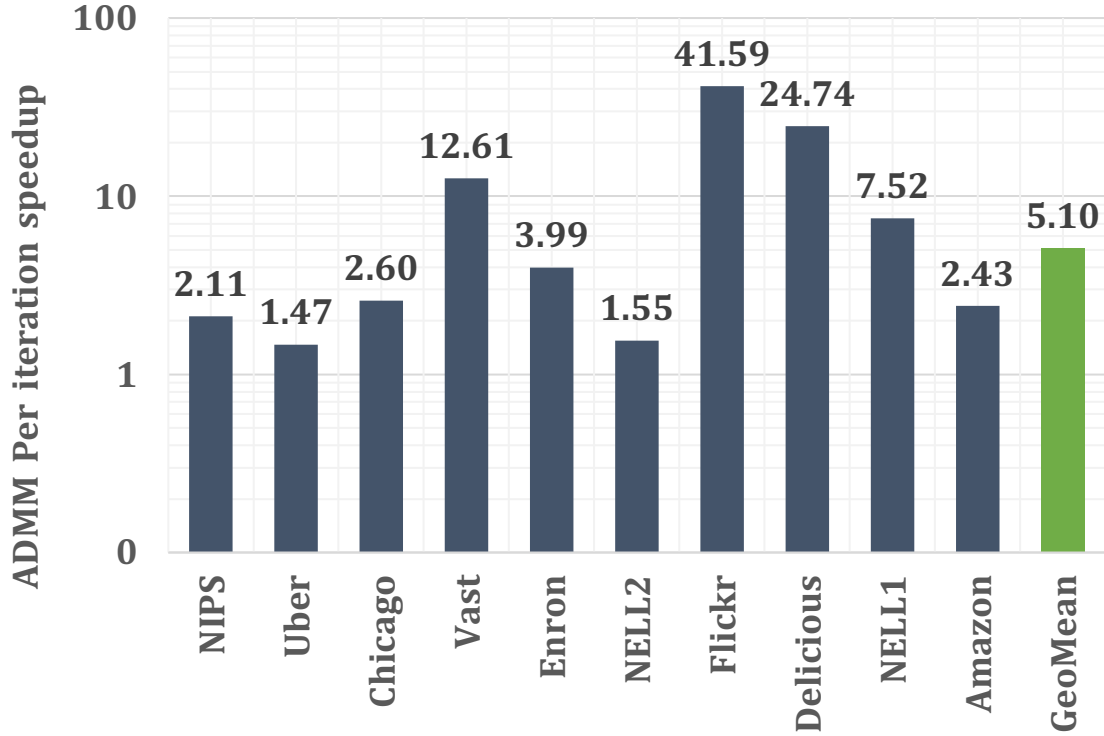


Figure 5.4. End-to-end per iteration speedup vs. SPLATT using ADMM update for rank $R = 32$ on A100 GPU.

Figures 5.4 and 5.5 shows the per-iteration end-to-end speedup of our framework against SPLATT using ADMM across 10 real-world datasets on the NVIDIA A100 and H100 GPUs, respectively. We achieve a geometric mean speedup of $5.10\times$ and $7.01\times$, and the speedup for each tensor ranges from $1.47\text{--}41.59\times$ and $1.22\text{--}58.05\times$ on the two GPUs, respectively. We achieve higher speedup on tensors with larger mode lengths, as having more elements to apply constraints to on the factor matrices corresponds to more parallel work that benefits from GPU’s massively parallel execution model.

The H100 GPU demonstrates higher performance compared to the A100 GPU,

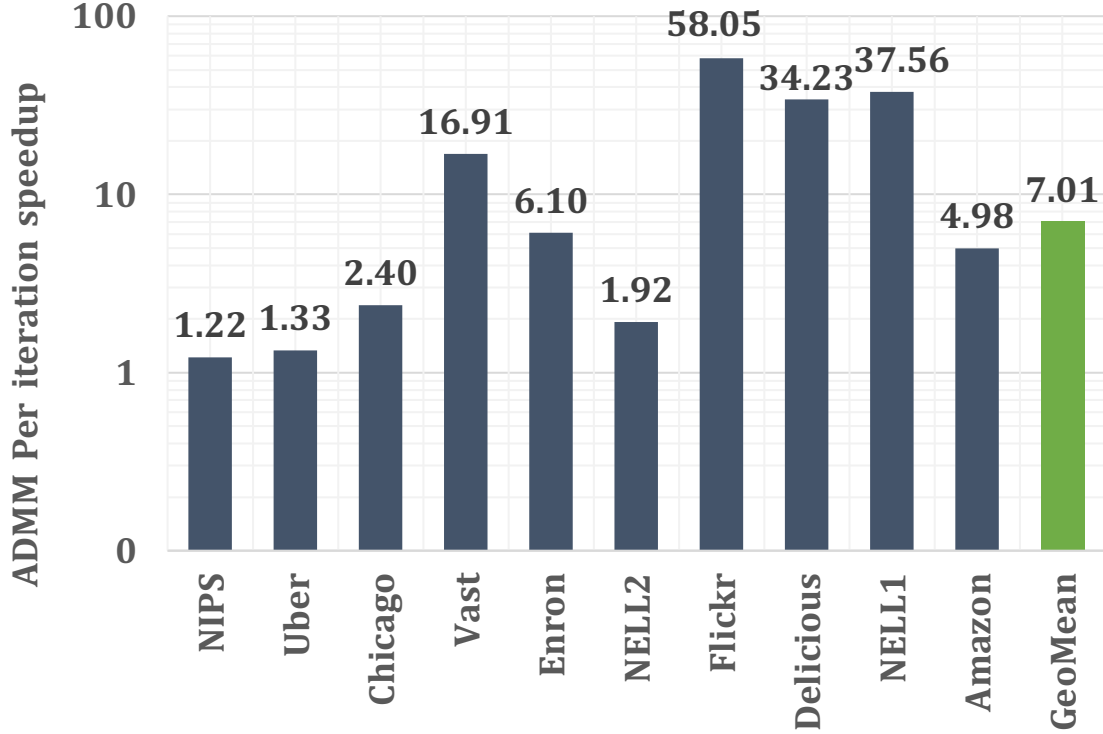


Figure 5.5. End-to-end per iteration speedup vs. SPLATT using ADMM update for rank $R = 32$ on H100 GPU.

despite the two devices having the same memory bandwidth. Our analysis of the ADMM algorithm (Section 6.2.2) reveals a low arithmetic intensity, which suggests that ADMM performance is bounded by the memory bandwidth of the device.

The MTTKRP algorithm demonstrates a similar behavior [16]. The higher performance on the H100 GPU comes from the higher overall cache size on the H100 GPU (20.3MB vs. 28.5MB L1D and 40MB vs. 50MB L2 on the A100 and H100, respectively). This result demonstrates the *importance of data reuse and caches* in the performance of cSTF.

Finally, we compare the speedup achieved by our GPU framework’s MTTKRP and ADMM kernels *against each other*. Figure 5.6 and 5.7 displays the speedup divided into two components— *MTTKRP* and *ADMM*—for the A100 and H100

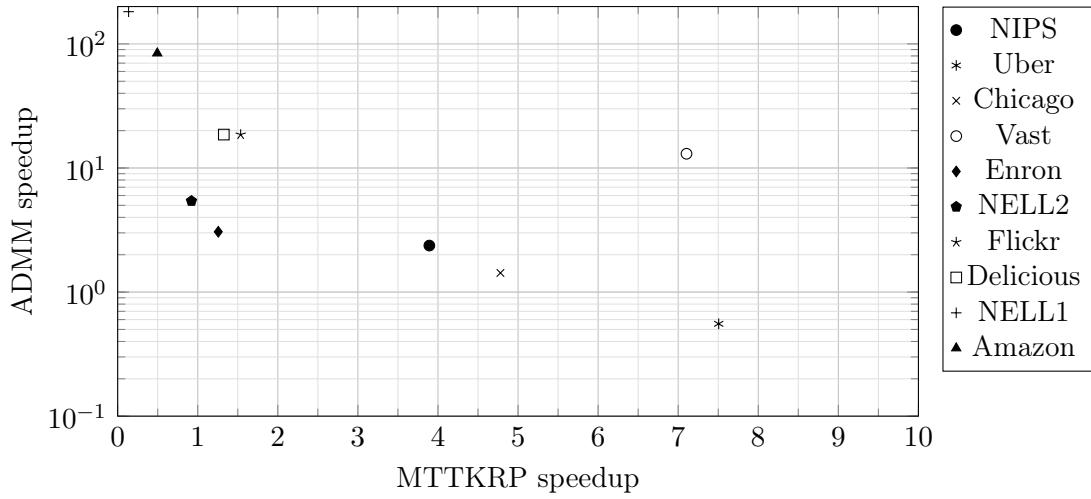


Figure 5.6. Comparison of speedup achieved for MTTKRP and ADMM across 10 real-world sparse tensors on the A100 GPU

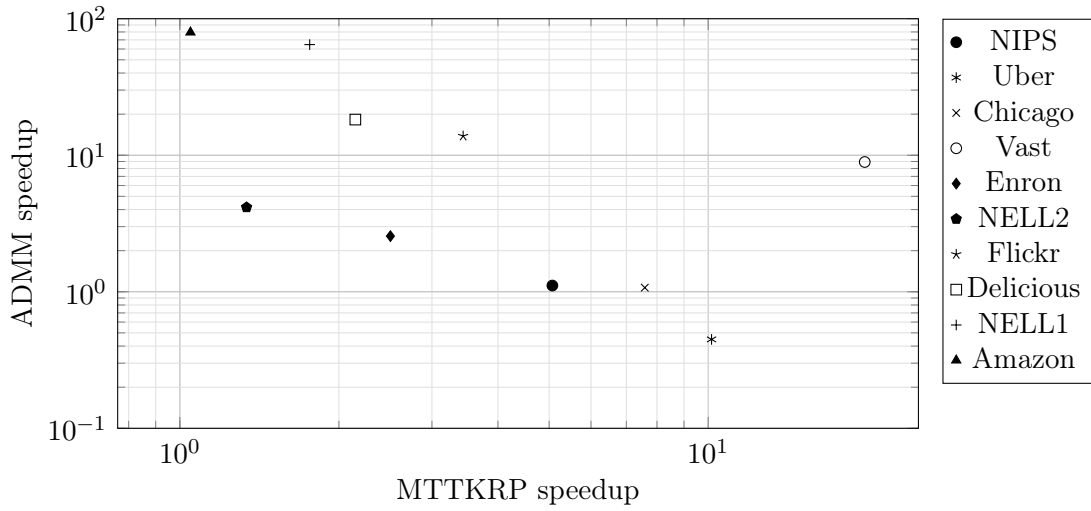


Figure 5.7. Comparison of speedup achieved for MTTKRP and ADMM across 10 real-world sparse tensors on the H100 GPU

GPUs, respectively. It can be seen that speedup for MTTKRP is *approximately indirectly proportional to the speedup* for ADMM, with the VAST tensor being the only exception. When the mode lengths are long (i.e., sparse tensors on the upper left corner), ADMM benefits more from GPU acceleration, as described in Section 5.4.2 (i.e., higher level of parallelism). However, longer mode lengths also result in higher sparsity, which lowers the performance achieved by the MTTKRP kernel, as higher sparsity generally lowers data reuse. Conversely, when the mode lengths are short (i.e., sparse tensors on the lower right corner), higher speedup is achieved for MTTKRP while lower speedup is achieved for ADMM. This results illustrates the *importance of optimizing both kernels*, as neglecting MTTKRP may result in no improvement in performance for sparse tensors with short mode lengths, and neglecting ADMM may result in no improvement in performance for sparse tensors with long mode lengths.

5.4.3 Accelerating additional non-negativity constraint algorithms

To demonstrate the flexibility of our GPU cSTF framework, we integrate two additional constraint update schemes—multiplicative update (MU) [50] and hierarchical alternating least square (HALS) [18] for non-negativity—and compare its performance against the CPU-based PLANC library that we modified to support sparse tensor factorization, as described in Section 5.3. Figures 5.8 and 5.9 show the achieved speedup. Our framework achieves geometric mean speedup of $6.42\times$ and $5.90\times$ for MU and HALS, respectively, on A100, and $8.89\times$ and $7.78\times$ on H100. This is comparable to the speedup achieved when using the ADMM update scheme.

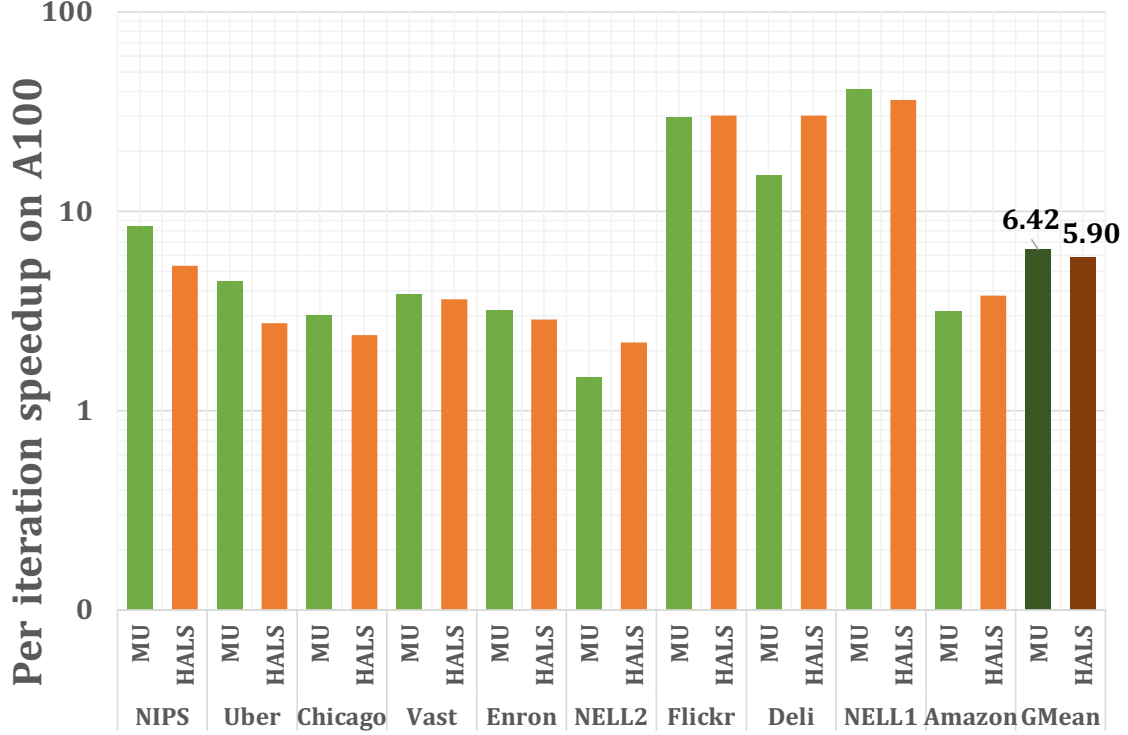


Figure 5.8. Speedup achieved on the A100 GPU by our GPU cSTF framework using the MU and HALS update schemes against PLANC, the state-of-the-art CPU-based cSTF library.

5.5 Conclusion

In summary, we have realized the following contributions that addresses our third research question: **RQ3:** With the advent of powerful GPUs, can we utilize these parallel computing architectures by implementing a fully GPU-resident constrained sparse tensor factorization framework?

1. We present the first GPU-accelerated framework for end-to-end constrained sparse tensor factorization (cSTF). We integrate the alternating optimization with alternating direction method of multipliers (AO-ADMM) update method, a fast and robust algorithm for applying constraints, to fully operate on the GPU.

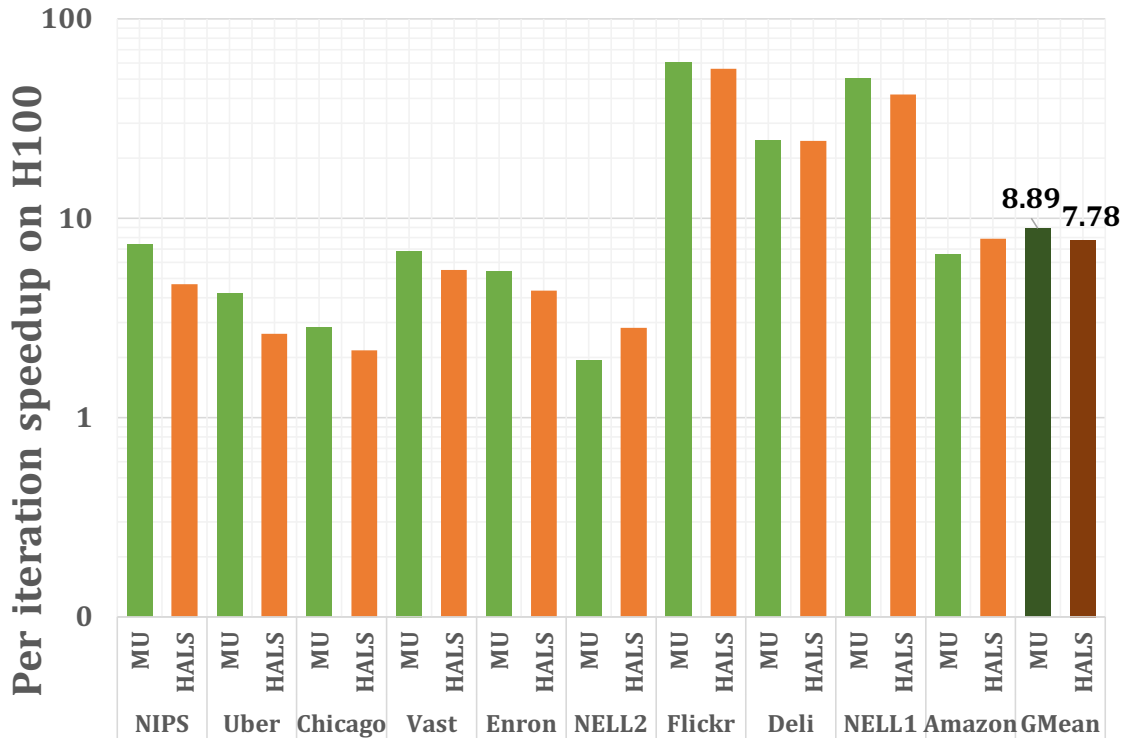


Figure 5.9. Speedup achieved on the H100 GPU by our GPU cSTF framework using the MU and HALS update schemes against PLANC, the state-of-the-art CPU-based cSTF library.

2. We analyze performance bottlenecks of the ADMM algorithm on GPUs and propose cuADMM, an optimized ADMM algorithm that eliminates redundant computation and data movement. cuADMM achieves a $1.8\times$ geometric mean speedup over generic ADMM on an NVIDIA H100 GPU across 10 real-world sparse tensors.
3. By offloading cSTF entirely to the GPU, our framework outperforms state-of-the-art CPU-based libraries, such as SPLATT [87]. We evaluate our framework on 10 real-world tensors and show that it achieves a geometric mean speedup (and maximum speedup) of $5.1\times$ ($41.59\times$) and $7.01\times$ ($58.05\times$) on the latest NVIDIA A100 and H100 GPUs, respectively.

4. We demonstrated our framework’s flexibility by incorporating two additional GPU-based non-negativity constraint algorithms, HALS and MU, which achieve significant speedups over their CPU counterparts.

CHAPTER 6

SCALABLE NONNEGATIVE LEAST SQUARES SOLVERS

This chapter contains previously published and unpublished work with the following co-authorship. Section 6.2 contains a part of the work from a previously published work at the 2021 International Parallel and Distributed Processing Symposium (IPDPS); specific co-authorship details for this publication are omitted here, as they were previously addressed in the opening of Chapter 3. Section 6.3 and Section 6.4 contain unpublished work under review. This work was done as a collaborative effort involving the University of Oregon and Oak Ridge National Laboratory. As the first author, I was responsible for the implementation, experimentation and the drafting of the manuscript. Co-authors Dr. Ramakrishnan Kannan, Dr. Piyush Sao, and Dr. Jee Choi provided technical guidance and assisted with the reviewing and editing of the manuscript.

6.1 Problem Statement

In prior chapters (Chapters 3 and 4), we focused on the optimization of the MTTKRP operation within the CP-ALS algorithm for sparse tensors. Orthogonal to the MTTKRP operation, in Chapter 5, we have addressed the drastic shift in the performance profile of CP-ALS for sparse tensors, where the constrained update process became the primary performance bottleneck. As such, we have presented an GPU-optimized ADMM implementation that addresses the *constrained update* portion of the CP-ALS algorithm, which enabled support for a fully offloaded GPU-resident constrained sparse tensor factorization framework.

For *constrained* tensor and matrix factorization algorithms, particularly those enforcing non-negativity, rely on specialized non-negative least squares (NNLS)

solvers to update the factor matrices. In this chapter, we will focus on two representative non-negative least squares solvers: ADMM and BPP. As we have already discussed the GPU optimized ADMM implementation in Chapter 5, we will focus on the optimization of the ADMM on multicore shared memory systems (Section 6.2) and a high throughput GPU realization of the BPP algorithm (Section 6.3).

Mathematically, both constrained and unconstrained approaches are fundamentally tasked with solving the optimization problem given by

$$\min_{\mathbf{X}} \frac{1}{2} \|\mathbf{AX} - \mathbf{B}\|_2^2, \quad (6.1)$$

where for non-negative constraint, we have $\mathbf{X} \geq 0$. It is crucial to note the distinction from the typical NNLS problem which solves for a single right-hand side (RHS). In the case for factorization algorithms, as we are solving a constrained optimization problem for the factor matrices, $\mathbf{X}$ and $\mathbf{B}$ are matrices which is effectively solving a NNLS problem for multiple right-hand sides (RHS) simultaneously.

Existing scalable matrix and tensor factorization softwares offer a wide suite of NNLS solvers at the users disposal, each designed with different priorities, such as scalability, accuracy, or robustness [18, 55, 41, 56, 64]. In Figure 6.1, we can see that out of the multiple NNLS solvers, alternating direction method of multipliers (ADMM) and block principal pivoting (BPP) are typically the best performing as they exhibit the lowest relative error over iterations, but this comes at a trade-off of being computationally expensive and not easily parallelizable, which can limit their application in large-scale factorization problems.

In this chapter, we will focus on a scalable and efficient implementation of two representative non-negative least squares solvers: ADMM and BPP.

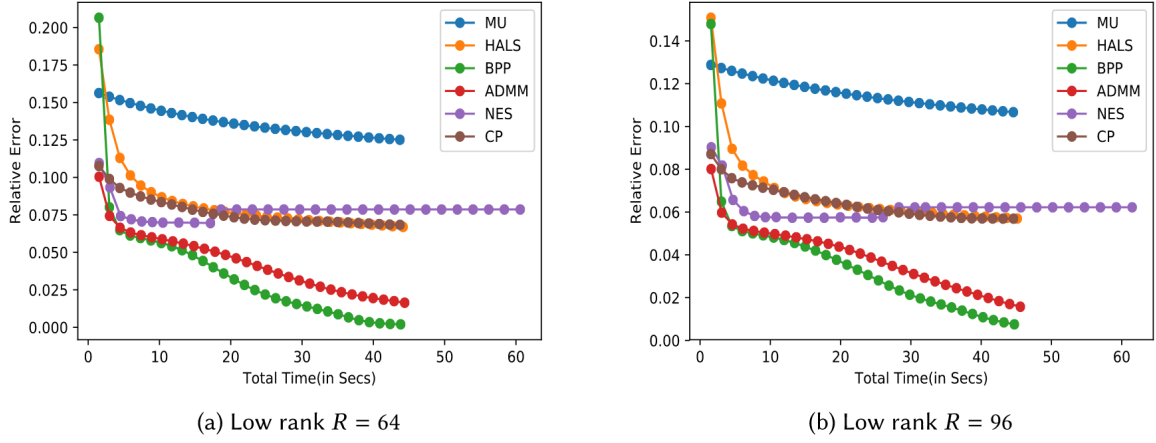


Figure 6.1. Motivational plot brought from [23]. Relative error over time comparison of updating algorithms on 4D Synthetic Low-rank Tensor of size $384 \times 384 \times 384 \times 384$ on $3 \times 3 \times 3 \times 3$ processor grid. MU: Multiplicative Update, HALS: Hierarchical Alternating Least Squares, ADMM: Alternating Direction Method of Multipliers, BPP: Block Principal Pivoting, CP: Unconstrained.

6.2 Accelerated ADMM on Multicore Shared Memory Systems

The Alternating Direction Method of Multipliers (ADMM) is a powerful algorithm used for solving optimization problems, particularly those with a linear constraint and a convex objective function that can be decomposed into two or more convex subproblems. As it can be parallelized efficiently, ADMM is particularly useful for solving problems with a large number of variables and constraints, which makes it a popular choice in non-negative factorization algorithms.

6.2.1 NNLS Formulation

ADMM solves the NNLS problem by reformulating it as the following constrained optimization problem:

$$\min_{\mathbf{X}, \mathbf{Z}} \frac{1}{2} \|\mathbf{A}\mathbf{X} - \mathbf{B}\|_F^2 + I_+(\mathbf{Z}) \quad \text{subject to} \quad \mathbf{X} - \mathbf{Z} = \mathbf{0} \quad (6.2)$$

The method is based on the Augmented Lagrangian Method but instead of

Algorithm 6 ADMM Algorithm

Input: $\mathbf{M}, \mathbf{S}, \mathbf{H}, \mathbf{U}, R$ $\triangleright \mathbf{M}$ is the MTTKRP matrix, R is the rank of the factors, $\mathbf{S}$ is the Hadamard of gram matrix, $\mathbf{H}, \mathbf{U}$ are the primal and dual variables respectively

- 1: Initialize $\mathbf{H}$ and $\mathbf{U}$
- 2: $\rho \leftarrow \text{trace}(\mathbf{S})/R$
- 3: Calculate $\mathbf{L}$ from the Cholesky decomp. of $\mathbf{S} + \rho\mathbf{I} = \mathbf{L}\mathbf{L}^T$
- 4: **repeat**
- 5: $\mathbf{H}_0 \leftarrow \mathbf{H}$
- 6: $\tilde{\mathbf{H}} \leftarrow (\mathbf{L}^T)^{-1}\mathbf{L}^{-1}(\mathbf{M} + \rho(\mathbf{H} + \mathbf{U}))^T$ $\triangleright$ Cholesky solve
- 7: $\mathbf{H} \leftarrow \arg \min_{\mathbf{H}} r(\mathbf{H}) + \frac{\rho}{2}\|\mathbf{H} - \tilde{\mathbf{H}}^T + \mathbf{U}\|_F^2$
- 8: $\mathbf{U} \leftarrow \mathbf{U} + \mathbf{H} - \tilde{\mathbf{H}}^T$
- 9: **until** $\|\mathbf{H} - \tilde{\mathbf{H}}\|_F^2/\|\mathbf{H}\|_F^2 < \epsilon$ **and** $\|\mathbf{H} - \mathbf{H}_0\|_F^2/\|\mathbf{U}\|_F^2 < \epsilon$
- 10: **return** $\mathbf{H}$

solving for joint minimization for $\mathbf{X}$ and $\mathbf{Z}$, it alternates between solving for $\mathbf{X}$ and $\mathbf{Z}$ while keeping the other fixed. In the alternating steps, ADMM solves the following subproblems:

$$\min_{\mathbf{X}} \frac{1}{2} \|\mathbf{A}\mathbf{X} - \mathbf{B}\|_F^2 + \frac{\rho}{2} \|\mathbf{X} - \mathbf{Z}\|_F^2 \quad (6.3)$$

$$\min_{\mathbf{Z}} I_+(\mathbf{Z}) + \frac{\rho}{2} \|\mathbf{X} - \mathbf{Z}\|_F^2 \quad (6.4)$$

where ρ is a penalty parameter. The pseudocode for ADMM in the context of CP-ALS is shown in Algorithm 6.

As we have addressed the GPU-optimized ADMM algorithm in Chapter 5, we will focus on the implementation details of ADMM on multicore shared memory systems and GPUs.

6.2.2 Computational and Memory Access Cost for ADMM

The *ADMM* algorithm is shown in Algorithm 6, and the associated computational and memory access cost for each operation are shown in Table 6.1. Notice that the expressions $(\Phi + \rho\mathbf{I})$ does not change within *ADMM*, and therefore is pre-computed outside and not counted (i.e., only the *Cholesky* operation for the *inverse*

Algorithm 7 Optimized ADMM Algorithm

Input: $\mathbf{A}, \Phi, \Psi$
Output: Updated factor matrix $\mathbf{A}$

```

1:  $\mathbf{U} \leftarrow 0$ 
2:  $\rho \leftarrow \text{trace}(\Phi)/K$ 
3:  $pr, pn, dr, dn \leftarrow 0$  ▷ Used for checking convergence
4:  $\mathbf{A}_0 \leftarrow \mathbf{A}$  ▷ Init
5: for each row-block  $B$  do
6:    $\tilde{\mathbf{A}}_B^T \leftarrow (\Phi_B + \rho \mathbf{I}_B)^{-1} (\Psi_B + \rho (\mathbf{A}_{0,B} + \mathbf{U}_B))^T$  ▷ Solve
7:    $\mathbf{A}_B \leftarrow \tilde{\mathbf{A}}_B - \mathbf{U}_B$ 
8:    $\mathbf{C}_{thr} \leftarrow \text{accum\_col\_norm}(\mathbf{A}_B)$  ▷ Per-thread
9: end for
10:  $\mathbf{C}_G \leftarrow \text{all-reduce}(\mathbf{C}_{thr})$ 
11: repeat
12:   for each row-block  $B$  do
13:      $\mathbf{A}_B \leftarrow \text{Proj}_C(\mathbf{A}_B, \mathbf{C}_G)$  ▷ Project
14:     for each element  $E$  in row-block  $B$  do
15:       register  $x \leftarrow \mathbf{A}_B[E]$ 
16:       register  $y \leftarrow x - \tilde{\mathbf{A}}_B[E]$ 
17:       register  $di \leftarrow \mathbf{U}_B[E] + y$ 
18:        $\mathbf{U}_B[E] \leftarrow di$  ▷ Update
19:        $pr \leftarrow pr + y \cdot y$  ▷  $\|\mathbf{A} - \tilde{\mathbf{A}}\|_F^2$ 
20:        $pn \leftarrow pn + x \cdot x$  ▷  $\|\mathbf{A}\|_F^2$ 
21:       register  $p \leftarrow x - \mathbf{A}_{0,B}[E]$ 
22:        $dr \leftarrow dr + p \cdot p$  ▷  $\|\mathbf{A} - \mathbf{A}_0\|_F^2$ 
23:        $dn \leftarrow dn + di \cdot di$  ▷  $\|\mathbf{U}\|_F^2$ 
24:        $\mathbf{A}_{0,B}[E] \leftarrow x$  ▷ Init
25:        $\tilde{\mathbf{A}}_B[E] \leftarrow (\Psi_B[E] + \rho(x + di))$ 
26:     end for
27:      $\tilde{\mathbf{A}}_B^T \leftarrow (\Phi_B + \rho \mathbf{I}_B)^{-1} \tilde{\mathbf{A}}_B^T$  ▷ Solve
28:      $\mathbf{A}_B \leftarrow \tilde{\mathbf{A}}_B - \mathbf{U}_B$ 
29:      $\mathbf{C}_{thr} \leftarrow \text{accum\_col\_norm}(\mathbf{A}_B)$ 
30:   end for
31:    $\mathbf{C}_G \leftarrow \text{all-reduce}(\mathbf{C}_{thr})$ 
32:    $\text{reduce}(pr, pn, dr, dn)$  ▷ OpenMP reduction
33: until  $pr/pn < \epsilon$  and  $dr/dn < \epsilon$ 
34: return  $\mathbf{A}$ 

```

is counted). The code is parallelized using *OpenMP* for most matrix operations in a fine-grained manner (i.e., one thread is assigned to each element), and parallelized over the columns for the column-wise norm calculation.

As it can be seen from Table 6.1, the arithmetic intensity of most operations in *ADMM* are < 0.125 (assuming double-precision 8-byte word), making them highly memory bandwidth-bound, according to the Roofline model [104]. Additionally,

many of the operations in Algorithm 6 are either *row-wise* or *element-wise* independent. The only exception is the *project* operation which involves column-wise norm calculation.

Table 6.1. Compute and memory costs for different ADMM operations.

Operation	Compute (flops)	Memory (words) (<i>Read</i>) + (<i>Write</i>)
<i>init</i>	0	$(IK) + (IK)$
<i>solve</i>	$3IK + 2IK^2$	$(4IK + K^2) + (2IK)$
<i>project</i>	$3IK + IK$	$(4IK) + (2IK)$
<i>update</i>	$2IK$	$(3IK) + (IK)$
<i>error</i>	$10IK$	$(4IK) + (0)$
Total	$19IK + 2IK^2$	$(16IK + K^2) + (6IK)$

6.2.3 Blocked and Fused ADMM

To address the memory-bandwidth bottleneck of ADMM, we apply the following optimizations:

1. *Data blocking*: divide the matrices into *blocks* of rows, and then parallelize over these blocks using *OpenMP* and vectorize the computation within each block,
2. *Operation fusion*: *fuse* matrix operations using registers to eliminate the *load* and *store* of intermediate results, and
3. Parallel reduction to calculate the column-wise *norm* across the blocks of rows.

Our optimized ADMM algorithm is shown in Algorithm 7. In this optimized version, we use *OpenMP* to parallelize over the row-blocks (i.e., one *OpenMP* thread is assigned to one block) in the *for* loops on line 5 and line 12. Computation inside

the *for* loop on line 14 are element-wise independent, so we vectorize the loop to take advantage of data level parallelism (DLP) and to improve the data access efficiency (i.e., fewer *load* instructions). The sequence of computation inside the innermost loop (i.e., the *init* $\rightarrow$ *solve* $\rightarrow$ *project* $\rightarrow$ *update* in the *repeat* loop in Algorithm 6) was rearranged to bring the *update*, *error*, *init*, and *solve* operations closer together so that registers can be used to hold the intermediate calculations rather than storing them in memory. Note that the *error* operation is interleaved with the other operations inside the inner loop.

This reduces the overall computational cost to $18IK + 2IK^2$ flops (vs. $19IK + 2IK^2$ flops), and data access to $15IK + K^2$ bytes (vs. $22IK + K^2$ bytes), which is more than a 30% reduction in data access. This is **not** counting the reduction in memory traffic due to blocking the matrices, which yields even more benefit.

Note that lines 7 and 8 (and lines 28 and 29) are also fused such that the each element of $\tilde{\mathbf{A}} - \mathbf{U}$ is stored in a register and used to calculate the norm (but we do not show this in Algorithm 7 to keep it concise). Also, we ignore the memory traffic associated with the per-thread array used to store the norms, as they generally tend to be very small in comparison to the matrices.

6.2.4 Performance Results for Optimized ADMM on Multicore Shared Memory Systems

Figure 6.3 compares the performance of our Blocked & Fused (**BF**) *ADMM* to the **baseline** *ADMM* implementation. As previously determined by our Roofline analysis (Section 6.2.2), the *ADMM* kernel is memory bandwidth-bound. This is why the execution time stops scaling for both implementations when the number of threads exceeds 14 for the *NIPS* dataset — there is already enough memory re-

quests to saturate the memory bandwidth. Our *cache blocking* and *operation fusion* optimizations, *while still memory bandwidth-bound*, reduces the overall memory traffic, leading to as much as $8.1\times$ speedup with 56 threads at rank-16. We omit the results for rank-64 and rank-128 as they show a similar trend as rank-16, with speedup peaking at $7.7\times$ and $4.2\times$, respectively.

The only anomaly occurs with 56 threads at rank-32, where we see a significantly higher speedup of $12.3\times$. This is likely due to our using the *minimum* execution time combined with particular poor execution for all 10 trials for the **baseline**, which also explains why the execution time *goes up* from 28 threads to 56 threads.

Figure 6.2 shows the speedup achieved by our Blocked & Fused *ADMM* for three different datasets using 56 threads. We see a similar trend as before, except with *Uber* showing lower speedup — this is due to the *factor matrices fitting in cache* (i.e., small dimension sizes for all modes), and as a result, the *data blocking* part of the optimization has no impact. The observed speedup can be mostly attributed to *operation fusion*.

6.3 Active-set Based Methods

Active-set based methods are a well-established approach for solving Nonnegative Least Squares (NNLS) problems [49, 10, 13, 100, 37, 21, 42, 61, 66, 20], often preferred over iterative projection-based methods due to their guaranteed convergence to an exact solution in finite steps [10]. These problems are critical components in applications like Non-negative Matrix or Tensor Factorization, which require solving for multiple right-hand sides (RHS) simultaneously. However, extending this framework to the multiple RHS setting introduces a fundamental parallelization challenge. The core computational bottleneck is solving very many (hun-

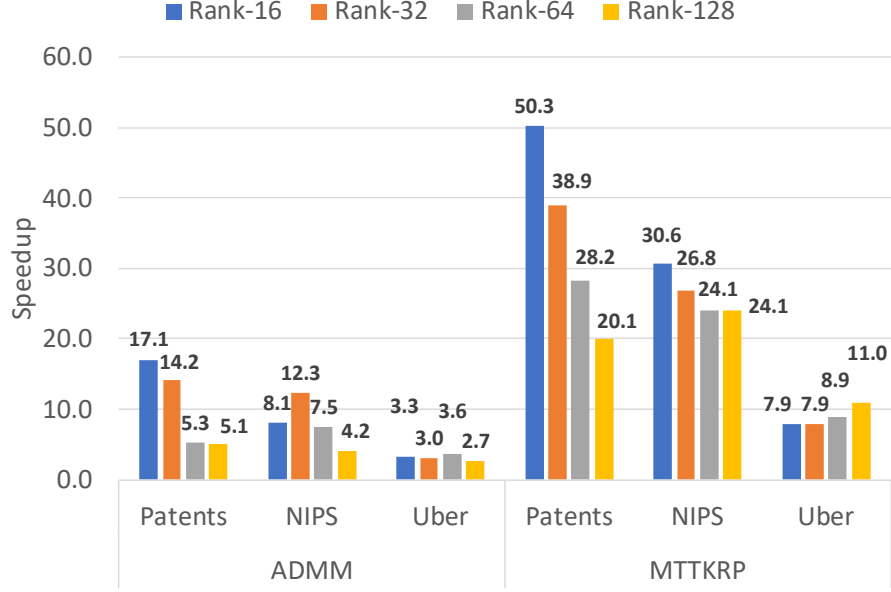


Figure 6.2. Speedup achieved by our Blocked & Fused *ADMM* and Hybrid Lock *MTTKRP* over the baseline for three datasets using 56 threads.

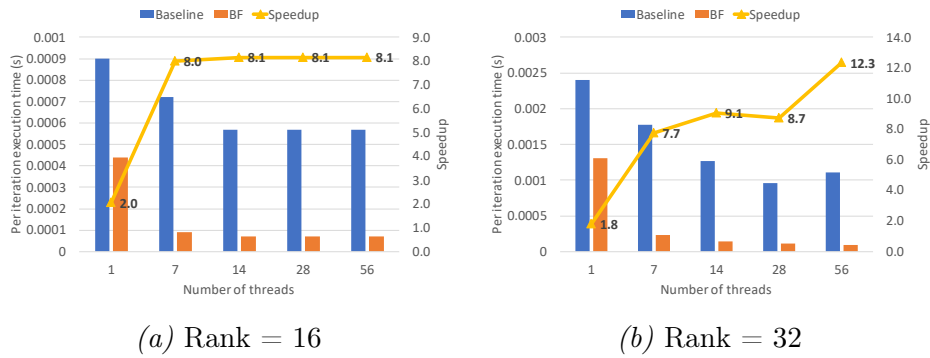


Figure 6.3. Comparison of Blocked & Fused *ADMM* kernel to the baseline on the NIPS dataset.

dreds of thousands) of small, structurally diverse linear systems (where each RHS corresponds to a single row of the factor matrices) that arise from a unique active set for each RHS. This irregularity prevents efficient batching, leading to extremely low hardware utilization on state-of-the-art GPUs.

6.3.1 NNLS Formulation

Unconstrained least squares problems $\min_{\mathbf{x}} \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_F^2$ have a direct solution through normal equations: $\mathbf{A}^T \mathbf{Ax} = \mathbf{A}^T \mathbf{b}$. However, non-negativity constraints $\mathbf{x} \geq 0$ eliminate this closed-form solution and requires an iterative method that respects constraint boundaries.

Active-set methods treat the constraint as a combinatorial problem. At the optimal solution, each variable is either zero (hitting the non-negativity boundary and therefore constraint is active) boundary) or positive (free from the constraint and therefore constraint is passive). The algorithm iteratively partitions variables into two sets:

- *Active set*: Variables constrained at zero
- *Passive set*: Variables free to take positive values

Active-set based methods then solves the unconstrained least squares subproblem over only the passive variables. This process continues until the correct partition of variables into active and passive sets satisfies the Karush-Kuhn-Tucker (KKT) conditions.

Consider $\mathbf{x} = [x_1, x_2, x_3]^T$ with unconstrained solution $\mathbf{x}_{unc} = [2.5, -1.2, 3.1]^T$. Here, $x_2 = -1.2$ violates non-negativity. An active-set method would:

1. Set the **active set** $\mathcal{A} = \{2\}$ ($x_2 = 0$)

2. Set the **passive set** $\mathcal{P} = \{1, 3\}$ ($x_1, x_3 > 0$)
3. Solve for passive variables: $\mathbf{A}_{:, \{1, 3\}} \mathbf{x}_{\{1, 3\}} = \mathbf{b}$
4. Check optimality: If $\nabla f_2 < 0$, move x_2 to passive set and repeat

This process systematically exchanges variables between sets until the KKT conditions are satisfied.

6.3.1.1 KKT Conditions for Active-set Based Methods

The Karush-Kuhn-Tucker (KKT) conditions formalize the optimality checks from the previous example. For a solution matrix $\mathbf{X}$ to be optimal, each element X_{ij} must satisfy three conditions with gradient $\mathbf{Y} = \mathbf{A}^T \mathbf{A} \mathbf{X} - \mathbf{A}^T \mathbf{B}$:

1. **Primal Feasibility:** $X_{ij} \geq 0$
2. **Stationarity:** $Y_{ij} = 0$ if $X_{ij} > 0$; $Y_{ij} \geq 0$ if $X_{ij} = 0$
3. **Complementary Slackness:** $X_{ij} Y_{ij} = 0$

These conditions guide the algorithm's decisions: variables with negative gradients move from active to passive sets, while variables that become negative move from passive to active sets.

6.3.2 The Block Principal Pivoting (BPP) Approach

Traditional active-set methods exchange only one variable per iteration. The Lawson-Hanson method [49] greedily selects variables based on gradient values. While this may exchange fewer total variables, single exchanges significantly increase iteration count.

Algorithm 8 NNLS with Block Principal Pivoting

Input: Matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, Matrix $\mathbf{B} \in \mathbb{R}^{m \times k}$

Output: Non-negative solution $\mathbf{X} \in \mathbb{R}^{n \times k}$

```

1: Initialize  $\mathbf{X} \leftarrow \mathbf{0}$ ,  $\mathbf{Y} \leftarrow -\mathbf{A}^T \mathbf{B}$ , Passive Set  $\mathcal{P} \leftarrow \text{false}$  for all entries
2: Initialize backup counters  $\mathbf{p}$  and infeasibility history  $\mathbf{n}_{\text{inf}}$  for each column.
3: Identify suboptimal columns  $\mathcal{C}_{\text{subopt}} \leftarrow \{j \mid \exists i : (Y_{ij} < 0 \wedge \neg P_{ij}) \vee (X_{ij} < 0 \wedge P_{ij})\}$ 
4: while  $\mathcal{C}_{\text{subopt}}$  is not empty do
5:   Check for maximum iteration count and terminate if exceeded.
6:   Partition  $\mathcal{C}_{\text{subopt}}$  into three sets based on backup rules
7:    $\mathcal{C}_1 \leftarrow \{j \in \mathcal{C}_{\text{subopt}} \mid \text{standard update rule applies}\}$ 
8:    $\mathcal{C}_2 \leftarrow \{j \in \mathcal{C}_{\text{subopt}} \mid \text{first backup rule applies}\}$ 
9:    $\mathcal{C}_3 \leftarrow \{j \in \mathcal{C}_{\text{subopt}} \mid \text{second backup rule applies}\}$ 
10:  for all columns  $j \in \mathcal{C}_1 \cup \mathcal{C}_2$  do
11:    Update passive set  $\mathcal{P}_j$  by moving all infeasible variables to the active set and all subop-
    timal variables to the passive set.
12:  end for
13:  for all columns  $j \in \mathcal{C}_3$  do
14:    Update passive set  $\mathcal{P}_j$  by flipping the status of a single variable.
15:  end for
16:  Solve normal equations for all suboptimal columns and update gradient:
17:   $\mathbf{X}_{\mathcal{C}_{\text{subopt}}} \leftarrow \text{MCNESolve}(\mathbf{A}^T \mathbf{A}, (\mathbf{A}^T \mathbf{B})_{\mathcal{C}_{\text{subopt}}}, \mathcal{P}_{\mathcal{C}_{\text{subopt}}})$ 
18:   $\mathbf{Y}_{\mathcal{C}_{\text{subopt}}} \leftarrow (\mathbf{A}^T \mathbf{A}) \mathbf{X}_{\mathcal{C}_{\text{subopt}}} - (\mathbf{A}^T \mathbf{B})_{\mathcal{C}_{\text{subopt}}}$ 
19:  Re-identify suboptimal columns  $\mathcal{C}_{\text{subopt}}$  based on updated  $\mathbf{X}$  and  $\mathbf{Y}$ .
20: end while
21:
22: return  $\mathbf{X}$ 

```

The Block Principal Pivoting (BPP) algorithm [37] offers a more efficient approach. BPP identifies all KKT-violating variables across columns and performs a "block pivot"—moving entire groups of variables in a single iteration. This block-wise strategy dramatically reduces required iterations but introduces the computational challenge of solving multiple mask-constrained normal equation systems simultaneously.

The BPP algorithm (Algorithm 8) repeats three steps until all columns satisfy KKT conditions:

1. **Identify KKT Violators (Line 3 and Line 19):** Find columns with violations: either infeasible (negative passive variable X_{ij}) or suboptimal (negative gradient component Y_{ij} for an active variable).

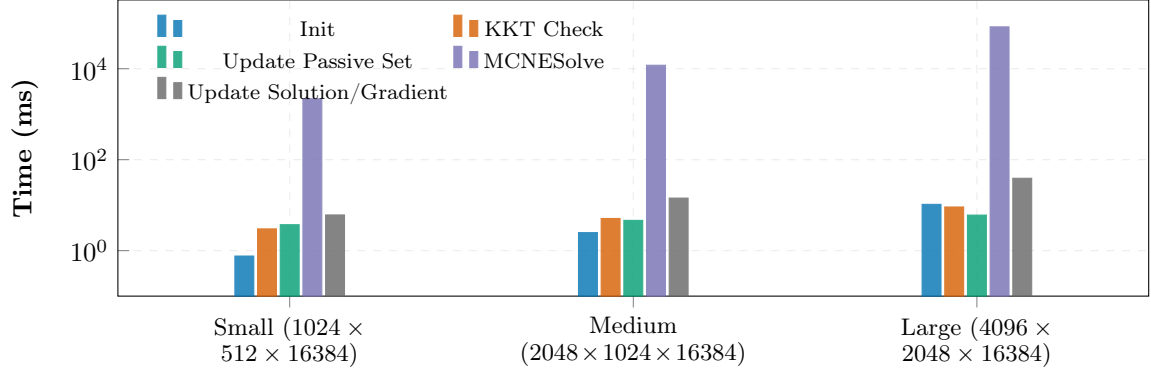


Figure 6.4. Execution time breakdown of Active-set NNLS Solver (Block Pivoting Principle (BPP) Method) for problems with varying scale. For all problem sizes, the Mask Constrained Normal Equation solve constitutes the primary computational bottleneck.

2. **Perform Block Pivot (Line 6 to 12):** Move all violating variables to the passive set simultaneously for all suboptimal columns. This accelerates the search for optimal active-passive partitioning.
3. **Solve Mask-Constrained Normal Equations (Line 17):** Solve normal equations for all suboptimal columns using the current passive set as mask constraints.

6.3.3 Performance Analysis of the Mask Constrained Normal Equation Solver

To reiterate, the objective of the active-set based methods is to find the optimal partition of variables into active and passive sets that satisfies the KKT conditions over a series of *search* iterations. Each iteration requires solving a subproblem for the current passive set. Figure 6.4 shows that solving this normal equation subproblem consumes the most computational time. We now examine this subproblem in detail.

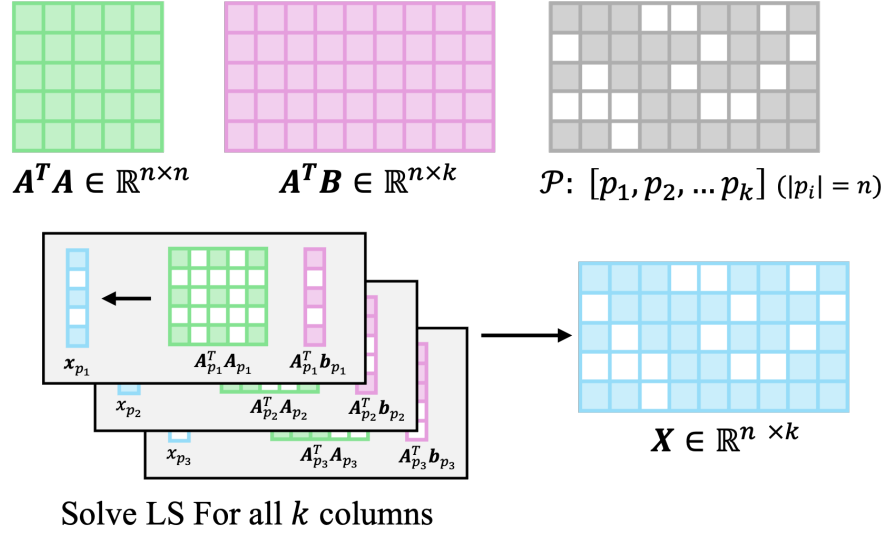


Figure 6.5. Illustration of the mask constrained normal equation problem. At each BPP iteration, the variables in the subsequent passive set for a given column i are found by solving the normal equations: $(A_{\mathcal{P}_i}^T A_{\mathcal{P}_i})(\mathbf{x}_i)_{\mathcal{P}_i} = A_{\mathcal{P}_i}^T \mathbf{b}_i$ for all i .

6.3.4 Problem Formulation

The Mask Constrained Normal Equation Solve (MCNESolve) in Algorithm 8 updates solution variables in the passive set for each search iteration (Figure 6.5). This function solves an equality-constrained least squares problem based on the current passive set partition. The output updates the solution $\mathbf{X}$ and gradient $\mathbf{Y}$.

For column j of solution matrix $\mathbf{X}$, let $\mathcal{P}_{:,j}$ be the passive set indices and $\mathcal{A}_j$ be the active set indices. The function solves:

$$\min_{\mathbf{x}_j} \frac{1}{2} \|\mathbf{A} \mathbf{x}_j - \mathbf{b}_j\|_2^2 \quad \text{subject to} \quad (\mathbf{x}_j)_{\mathcal{A}_j} = \mathbf{0}$$

This reduces to an unconstrained least squares problem over passive set columns:

$$\min_{(\mathbf{x}_j)_{\mathcal{P}_j}} \frac{1}{2} \|\mathbf{A}_{\mathcal{P}_j} (\mathbf{x}_j)_{\mathcal{P}_j} - \mathbf{b}_j\|_2^2$$

The corresponding normal equations use the principal submatrix $\mathbf{A}_{\mathcal{P}_j}^T \mathbf{A}_{\mathcal{P}_j}$ of the

global Gram matrix $\mathbf{A}^T \mathbf{A}$ indexed by $\mathcal{P}_j$:

$$(\mathbf{A}_{\mathcal{P}_j}^T \mathbf{A}_{\mathcal{P}_j})(\mathbf{x}_j)_{\mathcal{P}_j} = \mathbf{A}_{\mathcal{P}_j}^T \mathbf{b}_j.$$

6.3.4.1 Direct Factorization-based Solvers

Let $n_p = |\mathcal{P}_{:,j}|$ be the number of variables in the passive set for a given column j . This is a dense linear system of size $n_p \times n_p$, which we solve using Cholesky factorization. The cost includes factorizing $\mathbf{A}_{\mathcal{P}_j}^T \mathbf{A}_{\mathcal{P}_j}$ (approximately $\frac{1}{3}n_p^3$ FLOPs) and forward/backward substitutions (additional $2n_p^2$ FLOPs). Total FLOPs:

$$\text{FLOPs}_{\text{total}} \approx \frac{1}{3}n_p^3 + 2n_p^2 = O(n_p^3)$$

6.3.4.2 Conjugate Gradient Based Solvers

Alternatively, the normal equations can be solved using the Conjugate Gradient (CG) method. Let n_{cg} be the number of CG iterations required for convergence. Each CG iteration's most expensive operation is the matrix-vector product with the system matrix $\mathbf{A}_{\mathcal{P}_j}^T \mathbf{A}_{\mathcal{P}_j}$, requiring approximately $2n_p^2$ FLOPs. Each iteration includes vector operations (dot products and AXPY updates) contributing $O(n_p)$ FLOPs. Since the matrix-vector product dominates each iteration, total FLOPs:

$$\text{FLOPs}_{\text{total}} \approx n_{cg} \times (2n_p^2 + O(n_p)) = O(n_{cg}n_p^2)$$

This quadratic dependence on n_p offers significant advantage over cubic direct methods when $n_{cg} \ll n_p$.

Table 6.2. A comparison of FLOPS for different subproblem solvers

Solver type	Per iteration	#RHS	#it	Total FLOPS	$\text{FLOPS}_{\text{chol, trisolve}}$	FLOPS_{mv}	FLOPS_{mm}
Direct	$\frac{1}{3} n_p ^3 + 2 n_p ^2$	k	1	$k(\frac{1}{3} n_p ^3 + 2 n_p ^2)$	$k(\frac{1}{3} n_p ^3 + 2 n_p ^2)$	—	—
CG	$2 n_p ^2 + 12 n_p $	k	n_{cg}	$kn_{cg}(2 n_p ^2 + 12 n_p)$	—	$kn_{cg}(2 n_p ^2 + 12 n_p)$	—
BCG*	$2 n_p ^2 + 12 n_p $	k	n_{cg}	$kn_{cg}(2 n_p ^2 + 12 n_p)$	—	$kn_{cg} \times 12 n_p $	$kn_{cg}(2 n_p ^2)$
PBCG	$2 n_p ^2 + 12 n_p + 2 n_p ^2$	k	n_{pcg}	$kn_{pcg}(4 n_p ^2 + 12 n_p)$	$kn_{pcg}(2 n_p ^2)$	$kn_{pcg} \times 12 n_p $	$kn_{pcg}(2 n_p ^2)$

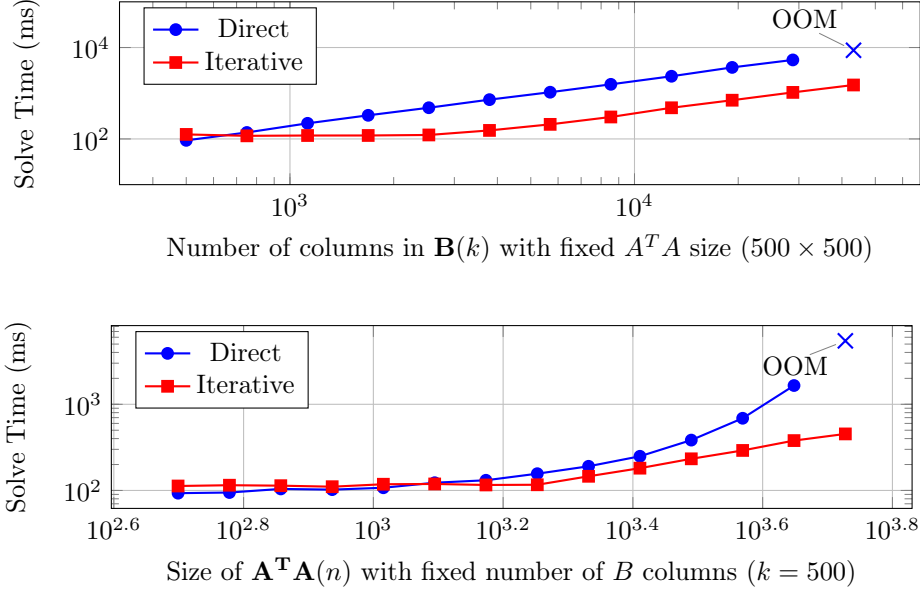


Figure 6.6. Direct vs. iterative solve times for mask constrained normal equations under various scaling conditions. Direct solvers are limited by memory because they must explicitly store principal submatrices of $\mathbf{A}^T \mathbf{A}$ for each column, causing them to fail as the problem size increase.

6.3.5 Limitations of Direct Factorization-based Solvers for Multiple RHS

Unlike unconstrained least squares (LS) problems that exploit BLAS Level 3 routines for multiple right-hand sides, Active-set based NNLS problems cannot benefit from these efficiencies. Direct factorization-based solvers must solve normal equations using principal submatrices of $\mathbf{A}^T \mathbf{A}$ for each column j in $\mathcal{P}$, requiring $O(n_p^3)$ operations per column (Illustration in Figure 6.5). Also, the optimization attempt to reduce the cost of Cholesky factorization by grouping identical columns [37, 42] fail at scale as the number of variables (i.e., k) grows. With 2^k possible passive set configurations for k variables, the probability of columns sharing identical sets vanishes as dimensions grow. Direct solvers also face memory bottlenecks from irregular gather-scatter operations when forming submatrices $\mathbf{A}_{\mathcal{P}_j}^T \mathbf{A}_{\mathcal{P}_j}$ for each pas-

sive set $\mathcal{P}_j$:

- **Memory fragmentation:** Non-contiguous access to extract principal submatrices
- **Cache inefficiency:** Repeated gathering of matrix entries across passive sets
- **Load imbalance:** Varying passive set sizes create irregular workloads

6.3.6 Limitations of Traditional Iterative Solvers for Constrained Problems

While Conjugate Gradient (CG) methods are asymptotically faster than direct solvers, traditional CG implementations face challenges in the constrained NNLS setting:

Poor GPU Utilization: Standard CG relies on Level-1/2 BLAS operations, which achieve poor performance on GPUs. CG iterations perform sparse dot products and matrix-vector multiplications that underutilize GPU resources.

Incompatibility with Block Methods: Block CG methods handle multiple right-hand sides using Level-3 BLAS operations for unconstrained problems. However, constrained NNLS requires different system matrices $\mathbf{A}_{\mathcal{P}_j}^T \mathbf{A}_{\mathcal{P}_j}$ for each column j based on unique passive sets. Traditional block Krylov methods [73] cannot handle this as they assume a shared system matrix.

Preconditioning Challenges: CG requires good preconditioning, yet dynamic passive sets complicate this:

- **Set-dependent structure:** Each passive set requires a different preconditioner

- **Amortization failure:** Construction costs cannot be amortized across columns with different passive sets
- **Update overhead:** Preconditioners must be recomputed as passive sets evolve during BPP iterations

These limitations show that existing solvers cannot efficiently handle mask-constrained normal equations in multiple-RHS NNLS problems.

6.4 FastBPP: Proposed Framework

While expressing the workload as matrix-matrix products and using efficient masking schemes provides a significant performance improvement potential over direct methods (Figure 6.6), the overall efficiency of an iterative solver is ultimately dictated by its convergence rate. A high number of CG iterations can easily diminish the benefits gained from superior data locality and parallelization over batched Cholesky solves. As such, minimizing the number of iterations for Block Masked CG (BCG) is paramount. We introduce a novel **Block Masked Preconditioned CG (PBCG)** subproblem solver designed specifically for this purpose. The proposed preconditioning strategy dramatically reduces the iteration count by clustering the system’s eigenvalues.

6.4.1 Block Masked Preconditioned CG

Preconditioning is a prevalent technique for accelerating the convergence of iterative methods [26, 28]. By effectively clustering its eigenvalue distribution, it enables the construction of a low-degree polynomial that serves as a closer approximation to the operator’s inverse on the subspace, leading to accelerated conver-

gence. Hypothetically, an ideal preconditioner would closely approximate the inverse of the system matrix, with a perfect preconditioner enabling convergence in a single iteration. We propose using the pre-computed Cholesky factorization of the global matrix (i.e., $\mathbf{H} = \mathbf{A}^T \mathbf{A}$) as a highly effective preconditioner for the series of related, but distinct, subproblems that arise in the mask constrained normal equation solve step.

6.4.1.1 Intuition

Mask constrained normal equations require solving linear systems of the form $\mathbf{H}_p \mathbf{x}_p = \mathbf{b}_p$, where the system matrix $\mathbf{H}_p$ is a principal submatrix of a larger, global symmetric positive definite (SPD) matrix $\mathbf{H}$. For each column, $\mathbf{H}_p$ is formed by only keeping rows and columns from $\mathbf{H}$ corresponding to passive set variables p . Since $\mathbf{H}_p$ structurally resembles $\mathbf{H}$, the global matrix inverse effectively preconditions $\mathbf{H}_p$. We perform a single Cholesky factorization of $\mathbf{H}$ to precondition each subproblem within a Preconditioned Conjugate Gradient (PCG) [26, 28] method. This approach solves each subproblem in iterations bounded by the number of active set variables.

6.4.2 Convergence Analysis

Given the global matrix $\mathbf{H} \in \mathbb{R}^{n \times n}$, consider a subproblem where the system matrix $\mathbf{H}_p \in \mathbb{R}^{(n-1) \times (n-1)}$ is a principal submatrix of $\mathbf{H}$ with one row and column removed. The corresponding column in this case would be a passive set of size $n - 1$ where only a single variable is an active variable. The system to solve is $\mathbf{H}_p \mathbf{x}_p = \mathbf{b}_p$.

We define our preconditioner, $\mathbf{M}_p^{-1}$, as the corresponding $(n - 1) \times (n - 1)$ principal submatrix of $\mathbf{H}^{-1}$. The relationship between the preconditioner and the

true inverse of the submatrix is given by the Woodbury matrix identity [26], which simplifies to: $\mathbf{M}_p^{-1} = \mathbf{H}_p^{-1} + \mathbf{u}\mathbf{v}^T$ where $\mathbf{u}, \mathbf{v} \in \mathbb{R}^{n-1}$, making $\mathbf{u}\mathbf{v}^T$ a rank-1 matrix. The preconditioned matrix, $\mathbf{K}$, is therefore:

$$\mathbf{K} = \mathbf{M}_p^{-1}\mathbf{H}_p = (\mathbf{H}_p^{-1} + \mathbf{u}\mathbf{v}^T)\mathbf{H}_p = \mathbf{I} + \mathbf{u}(\mathbf{v}^T\mathbf{H}_p) = \mathbf{I} + \mathbf{u}\mathbf{w}^T.$$

As $\mathbf{K}$ is an identity plus a rank-1 matrix, $\mathbf{K}$ has at most two distinct eigenvalues and CG method is guaranteed to converge in at most 2 iterations.

This analysis can extend to the general case where $\mathbf{H}_p \in \mathbb{R}^{(n-k) \times (n-k)}$ is formed by removing k variables. A generalization of the Woodbury identity generalizes to show that the preconditioner $\mathbf{M}_p^{-1}$ (the $(n-k) \times (n-k)$ principal submatrix of $\mathbf{H}^{-1}$) is related to the true inverse by a rank- k update: $\mathbf{M}_p^{-1} = \mathbf{H}_p^{-1} + \mathbf{R}_k$ where $\text{rank}(\mathbf{R}_k) \leq k$. The preconditioned matrix $\mathbf{K}$ is then:

$$\mathbf{K} = \mathbf{M}_p^{-1}\mathbf{H}_p = (\mathbf{H}_p^{-1} + \mathbf{R}_k)\mathbf{H}_p = \mathbf{I} + \mathbf{R}_k\mathbf{H}_p = \mathbf{I} + \mathbf{R}'_k$$

where $\text{rank}(\mathbf{R}'_k) \leq k$, where it becomes the identity plus a low-rank matrix. This structure ensures that the preconditioned matrix $\mathbf{K}$ has at most k eigenvalues different from 1, resulting in at most $k+1$ distinct eigenvalues. and is guaranteed to converge in at most $k+1$ iterations.

While Cholesky downdating may seem like a viable alternative, its application in this context has significant performance challenges in a massively parallel setting [9]. To compute the Cholesky factors of a submatrix where k variables have been removed, one must apply a series of k rank-1 downdates. This process introduces a computational dependency that is inherently sequential and inefficient for parallel execution.

While more advanced rank- k (or block) downdating methods exist, they suffer from similar scalability issues. These algorithms typically rely on a series of orthogonal transformations, such as Givens rotations, to zero out the contributions of the

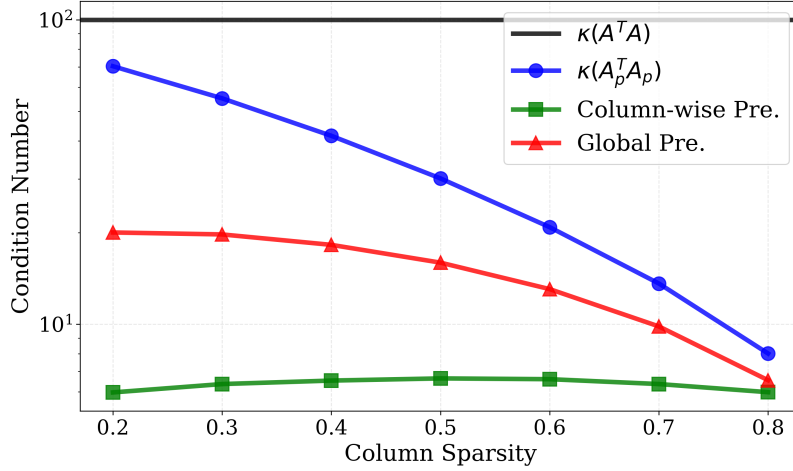


Figure 6.7. Preconditioning effectiveness for two distinct preconditioning strategies. Column-wise preconditioning offers better conditioning, the global preconditioner provides improvement over the original subproblem $\kappa(A_p^T A_p)$ while being more parallel execution-friendly.

removed variables. More critically, their formulation often requires solving an auxiliary $k \times k$ linear system [7]. For a multiple-RHS problem with diverse passive sets, this would reintroduce a batch of small, irregular Cholesky factorizations—precisely the kind of parallelism-averse workload our iterative approach is designed to avoid.

6.4.3 Efficacy of Global Preconditioning for PBCG

We explored two strategies for using a global Cholesky factor of $\mathbf{L}\mathbf{L}^T = \mathbf{A}^T\mathbf{A}$ as a preconditioner for column-specific subproblems.

The first strategy extracts relevant submatrices from $\mathbf{L}$ to form local preconditioners $\mathbf{L}_p$. While this offers superior conditioning, it requires computing and storing unique inverses for each subproblem, mirroring the high memory costs of a direct batched Cholesky solver.

We adopt the second strategy, applying the full inverse of the global matrix $(\mathbf{A}^T\mathbf{A})^{-1}$ to subproblems embedded within the original matrix dimensions. As

shown in Figure 6.7, this method yields slightly higher condition numbers than the first strategy, but preserves batched processing and consistent memory usage, making it more suitable for parallel GPU architectures. The computational efficiency gained outweighs the modest decrease in conditioning.

Our resulting Block Masked Preconditioned Conjugate Gradient (PBCG) solver is particularly effective for columns with dense passive sets, significantly reducing CG iterations compared to BCG. By leveraging global factor information, this method combines performance and numerical robustness within our block principal pivoting framework.

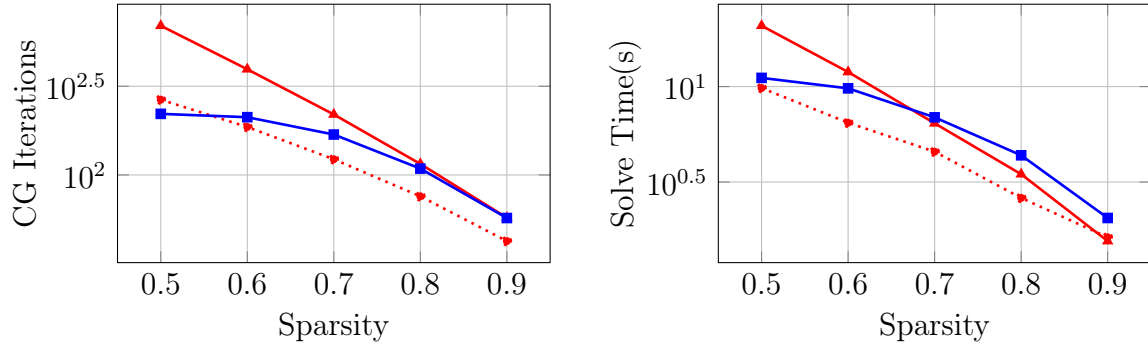


Figure 6.8. Performance comparison of Block Masked CG based solvers for high column sparsity (>0.5). For denser columns, PBCG always outperforms BCG based solvers.

6.4.4 Sparsity-based Adaptive Block Masked CG

While PBCG is effective when k is small (i.e., passive set columns are dense), its theoretical guarantee of $k+1$ iterations degrades as k grows. When k approaches n , the preconditioner becomes less effective.

We therefore adopt an adaptive strategy within our block pivoting framework, monitoring the number of active variables per column, $n - k$:

- If $n - k$ is small (large k), we use standard Block Masked Conjugate Gradient (BCG).
- If $n - k$ is large (small k), we use PBCG, as the global preconditioner closely approximates the subproblem.

This hybrid approach leverages PBCG’s rapid convergence for nearly-dense problems and BCG’s reliability for sparser cases. Figure 6.8 validates this strategy by comparing CG iterations and solve times across varying sparsities.

We also realized BCG with column masking, which deactivates converged columns during CG iterations. This technique benefits BCG, whose columns converge at different rates, by progressively reducing the problem size. However, it offers little advantage for PBCG since its columns converge uniformly and the masking overhead negates any gains.

6.4.5 Maintaining Optimality with Direct Solver Verification

Block Masked CG-based subproblem solvers offer computational efficiency but produce approximate solutions, potentially compromising the numerical accuracy of active-set NNLS solvers. To address this, we introduce a selective verification mechanism for recently converged columns. When a column satisfies the KKT conditions, it undergoes verification using a direct solver. This approach maintains computational efficiency while ensuring solution accuracy equivalent to purely direct methods, preserving the theoretical optimality guarantees of the active-set based method.

This hybrid strategy selectively uses direct subproblem solvers, introducing a controlled computational overhead. While direct solvers are used for verification, the overall cost remains lower than a purely direct approach. Exact solutions are

required only once per column, when it reaches its optimal configuration. The majority of the computational work during active-set exploration benefits from efficient iterative subproblem solvers. This strategic blend of iterative exploration and direct verification forms the basis of our final algorithm, FastBPP, which is detailed in Algorithm 9.

Algorithm 9 FastBPP

Input: Matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, Matrix $\mathbf{B} \in \mathbb{R}^{m \times k}$

Output: Non-negative solution $\mathbf{X} \in \mathbb{R}^{n \times k}$

```

1: Initialize  $\mathbf{X} \leftarrow \mathbf{0}$ ,  $\mathbf{Y} \leftarrow -\mathbf{A}^T \mathbf{B}$ , Passive Set  $\mathcal{P} \leftarrow \text{false}$  for all entries
2: Initialize set of pending verification columns  $\mathcal{C}_{\text{pending}} \leftarrow \emptyset$ .
3: Initialize set of previously suboptimal columns  $\mathcal{C}_{\text{prev\_subopt}} \leftarrow \{1, \dots, k\}$ .
4: for  $iter = 1 \rightarrow \max\_iter$  do
5:   if  $\mathcal{C}_{\text{pending}}$  is not empty then
6:      $\mathbf{X}_{\mathcal{C}_{\text{pending}}} \leftarrow \text{DirectSolver}(\mathbf{A}^T \mathbf{A}, (\mathbf{A}^T \mathbf{B})_{\mathcal{C}_{\text{pending}}}, \mathcal{P}_{\mathcal{C}_{\text{pending}}})$ 
7:      $\mathbf{Y}_{\mathcal{C}_{\text{pending}}} \leftarrow (\mathbf{A}^T \mathbf{A}) \mathbf{X}_{\mathcal{C}_{\text{pending}}} - (\mathbf{A}^T \mathbf{B})_{\mathcal{C}_{\text{pending}}}$ 
8:   end if
9:   Identify current suboptimal columns  $\mathcal{C}_{\text{subopt}}$  based on KKT violations.
10:  Identify transitioning columns  $\mathcal{C}_{\text{trans}} \leftarrow \mathcal{C}_{\text{prev\_subopt}} \setminus \mathcal{C}_{\text{subopt}}$ .
11:  Set  $\mathcal{C}_{\text{pending}} \leftarrow \mathcal{C}_{\text{trans}}$  for verification in the next iteration.
12:  Set  $\mathcal{C}_{\text{prev\_subopt}} \leftarrow \mathcal{C}_{\text{subopt}}$ .
13:  if  $\mathcal{C}_{\text{subopt}}$  is empty then
14:    break ▷ All columns are either optimal or pending verification.
15:  end if
16:  Update passive set  $\mathcal{P}_{\mathcal{C}_{\text{subopt}}}$  using block pivoting and backup rules.
17:  if adaptive criteria met (e.g., few active variables) then
18:     $\mathbf{X}_{\mathcal{C}_{\text{subopt}}} \leftarrow \text{PreconditionedBlockCG}(\mathbf{A}^T \mathbf{A}, (\mathbf{A}^T \mathbf{B})_{\mathcal{C}_{\text{subopt}}}, \mathcal{P}_{\mathcal{C}_{\text{subopt}}})$ 
19:  else
20:     $\mathbf{X}_{\mathcal{C}_{\text{subopt}}} \leftarrow \text{BlockCG}(\mathbf{A}^T \mathbf{A}, (\mathbf{A}^T \mathbf{B})_{\mathcal{C}_{\text{subopt}}}, \mathcal{P}_{\mathcal{C}_{\text{subopt}}})$ 
21:  end if
22:   $\mathbf{Y}_{\mathcal{C}_{\text{subopt}}} \leftarrow (\mathbf{A}^T \mathbf{A}) \mathbf{X}_{\mathcal{C}_{\text{subopt}}} - (\mathbf{A}^T \mathbf{B})_{\mathcal{C}_{\text{subopt}}}$ 
23: end for
24: if  $\mathcal{C}_{\text{pending}}$  is not empty then ▷ Verify any columns that became optimal on the last iteration.
25:    $\mathbf{X}_{\mathcal{C}_{\text{pending}}} \leftarrow \text{DirectSolver}(\mathbf{A}^T \mathbf{A}, (\mathbf{A}^T \mathbf{B})_{\mathcal{C}_{\text{pending}}}, \mathcal{P}_{\mathcal{C}_{\text{pending}}})$ 
26:   Update corresponding columns in global solution  $\mathbf{X}$ .
27: end if
28:
29: return  $\mathbf{X}$ 

```

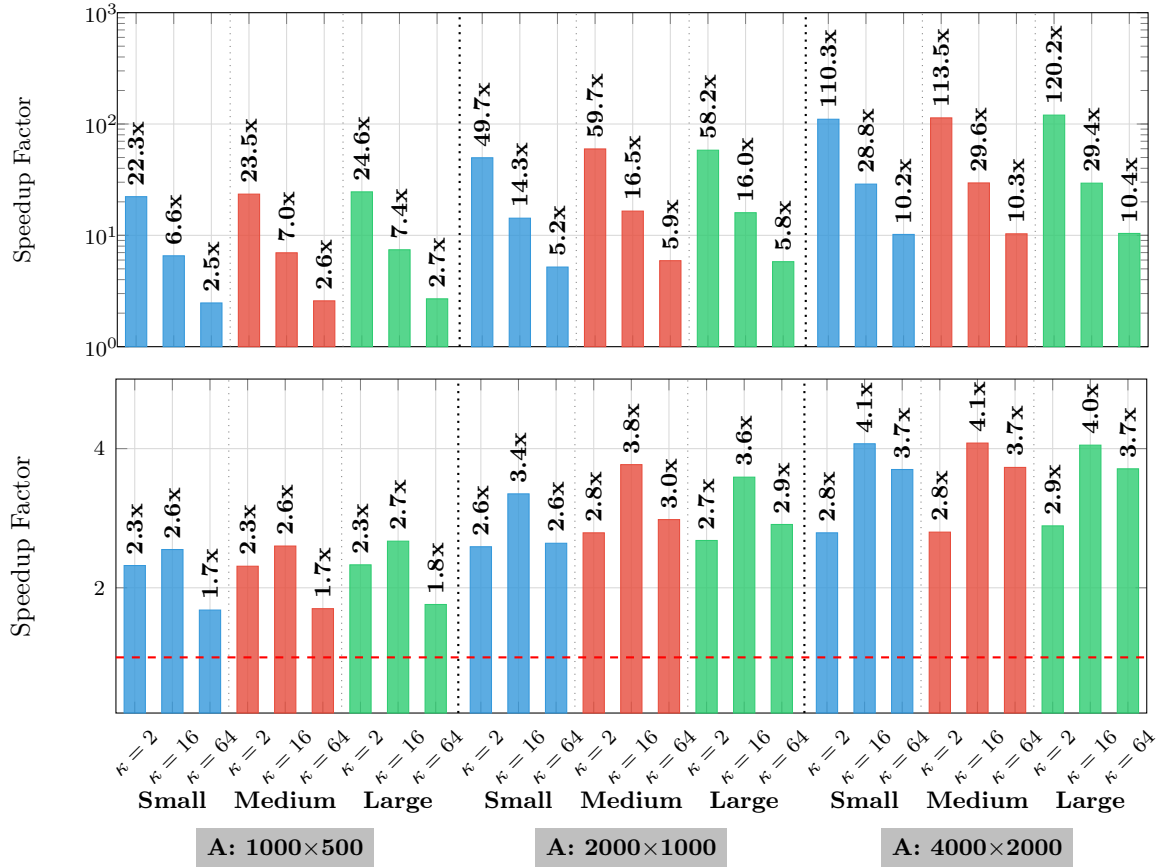


Figure 6.9. The top and bottom plots show the speedup of **FastBPP** and **FastBPP-V**, respectively, over a baseline method. Each bar represents a unique combination of matrix size, problem size, and condition number. The plot is organized in three major groups corresponding to different **A** matrix dimensions: 1000×500 , 2000×1000 , and 4000×2000 . Within each matrix size group, there are three problem size subgroups (Small, Medium, Large) based on the number of right-hand sides (25,000, 50,000, and 100,000, respectively). Each subgroup contains three bars showing results for various condition numbers (κ).

6.4.6 Performance Results for Block Principal Pivoting on Massively Parallel GPU Systems

6.4.6.1 Experimental Setup and Configuration

All experiments were conducted on an NVIDIA H100 PCIe GPU with 80 GB memory, using CUDA 12.8, PyTorch 2.7.1, and Python 3.12. The system was equipped

with 503 GB of system RAM to accommodate the large-scale problem instances.

We generate synthetic large-scale test problems across three matrix dimensions: $\mathbf{A} \in \mathbb{R}^{1000 \times 500}$, $\mathbf{A} \in \mathbb{R}^{2000 \times 1000}$, and $\mathbf{A} \in \mathbb{R}^{4000 \times 2000}$, with corresponding multiple right-hand sides $\mathbf{B} \in \mathbb{R}^{m \times k}$, where $k \in \{25000, 50000, 100000\}$. To assess solver robustness across varying numerical challenges, we control the condition number of the Gram matrix, $\kappa(\mathbf{A}^T \mathbf{A})$, across seven values $\{4, 16, 64, 256, 1024, 4096, 16384\}$. The ground-truth solutions, $\mathbf{X}_{\text{true}}$, are generated with three distinct sparsity levels: $s \in \{0.2, 0.5, 0.8\}$, where s represents the fraction of zero elements in the solution matrix. Using five different random seeds, we generate a comprehensive test suite of 945 problems each consisting of the synthetic matrix $\mathbf{A}$ and ground truth solution $\mathbf{X}_{\text{true}}$, ensuring statistical reliability across all experimental conditions. As our contribution is an acceleration of the established Block Pivoting Principle (BPP) method, this approach allows us to systematically vary problem characteristics and isolate the impact of our work.

6.4.6.2 NNLS Solver Configurations

We evaluate three distinct, fully GPU-resident solver configurations. Our baseline, **BPP-BC**, is a BPP implementation utilizing batched Cholesky factorization for the subproblem solve. The proposed **FastBPP** solver is a hybrid iterative method that dynamically switches between a Block Masked Preconditioned Conjugate Gradient (PBCG) for dense passive set columns and a Block Masked Conjugate Gradient with Column Masking (BCG) for sparse ones. The third configuration, **FastBPP-V** extends **FastBPP** by adding a final verification step where a direct solver (batched Cholesky) recalculates the solution for any column transitioning to an optimal state, ensuring numerical exactness while retaining the benefits of the iterative solver.

Table 6.3. Performance comparison of **FastBPP** variants

A Size	RHS	Solver Type	Solve Time (s)						
			$\kappa=2$	$\kappa=4$	$\kappa=8$	$\kappa=16$	$\kappa=32$	$\kappa=64$	$\kappa=128$
1000×500	25K	BPP-BC	2.37	2.87	3.40	3.83	4.22	4.74	5.14
		FastBPP	0.11 (22 $\times$)	0.18 (16 $\times$)	0.33 (10 $\times$)	0.58 (7 $\times$)	1.04 (4 $\times$)	1.92 (3 $\times$)	5.00 (1 $\times$)
		FastBPP-V	1.02 (2 $\times$)	1.11 (3 $\times$)	1.26 (3 $\times$)	1.50 (3 $\times$)	1.98 (2 $\times$)	2.82 (2 $\times$)	5.95 (1 $\times$)
	50K	BPP-BC	4.52	5.60	6.51	7.45	8.32	9.18	10.10
		FastBPP	0.19 (24 $\times$)	0.32 (17 $\times$)	0.58 (11 $\times$)	1.07 (7 $\times$)	1.88 (4 $\times$)	3.56 (3 $\times$)	8.31 (1 $\times$)
		FastBPP-V	1.95 (2 $\times$)	2.11 (3 $\times$)	2.39 (3 $\times$)	2.86 (3 $\times$)	3.67 (2 $\times$)	5.41 (2 $\times$)	10.12 (1 $\times$)
	100K	BPP-BC	8.99	11.12	13.04	14.87	16.63	18.49	20.37
		FastBPP	0.37 (25 $\times$)	0.62 (18 $\times$)	1.11 (12 $\times$)	2.00 (7 $\times$)	3.64 (5 $\times$)	6.88 (3 $\times$)	14.89 (1 $\times$)
		FastBPP-V	3.85 (2 $\times$)	4.21 (3 $\times$)	4.71 (3 $\times$)	5.56 (3 $\times$)	7.26 (2 $\times$)	10.53 (2 $\times$)	18.71 (1 $\times$)
2000×1000	25K	BPP-BC	12.84	15.88	18.41	20.97	23.51	25.93	28.39
		FastBPP	0.26 (50 $\times$)	0.42 (38 $\times$)	0.80 (23 $\times$)	1.47 (14 $\times$)	2.63 (9 $\times$)	5.00 (5 $\times$)	9.66 (3 $\times$)
		FastBPP-V	4.95 (3 $\times$)	5.18 (3 $\times$)	5.57 (3 $\times$)	6.26 (3 $\times$)	7.44 (3 $\times$)	9.81 (3 $\times$)	14.48 (2 $\times$)
	50K	BPP-BC	27.71	34.16	40.53	46.02	51.95	57.35	63.42
		FastBPP	0.46 (60 $\times$)	0.85 (40 $\times$)	1.52 (27 $\times$)	2.79 (17 $\times$)	5.12 (10 $\times$)	9.68 (6 $\times$)	18.67 (3 $\times$)
		FastBPP-V	9.93 (3 $\times$)	10.25 (3 $\times$)	10.93 (4 $\times$)	12.21 (4 $\times$)	14.66 (4 $\times$)	19.24 (3 $\times$)	28.19 (2 $\times$)
	100K	BPP-BC	52.80	65.18	76.90	87.74	100.62	111.63	124.56
		FastBPP	0.91 (58 $\times$)	1.63 (40 $\times$)	2.98 (26 $\times$)	5.49 (16 $\times$)	10.30 (10 $\times$)	19.25 (6 $\times$)	37.12 (3 $\times$)
		FastBPP-V	19.73 (3 $\times$)	20.45 (3 $\times$)	21.83 (4 $\times$)	24.42 (4 $\times$)	29.39 (3 $\times$)	38.32 (3 $\times$)	56.11 (2 $\times$)
4000×2000	25K	BPP-BC	79.08	98.56	114.63	130.18	144.66	159.63	174.67
		FastBPP	0.72 (110 $\times$)	1.34 (74 $\times$)	2.43 (47 $\times$)	4.51 (29 $\times$)	8.28 (18 $\times$)	15.66 (10 $\times$)	30.10 (6 $\times$)
		FastBPP-V	28.34 (3 $\times$)	28.94 (3 $\times$)	30.06 (4 $\times$)	32.02 (4 $\times$)	35.93 (4 $\times$)	43.19 (4 $\times$)	57.68 (3 $\times$)
	50K	BPP-BC	159.51	198.88	231.74	262.94	292.77	323.93	354.28
		FastBPP	1.41 (114 $\times$)	2.54 (78 $\times$)	4.75 (49 $\times$)	8.90 (30 $\times$)	16.90 (17 $\times$)	31.40 (10 $\times$)	59.53 (6 $\times$)
		FastBPP-V	57.04 (3 $\times$)	58.14 (3 $\times$)	60.34 (4 $\times$)	64.46 (4 $\times$)	72.39 (4 $\times$)	86.94 (4 $\times$)	114.90 (3 $\times$)
	100K	BPP-BC	331.58	399.10	464.25	527.07	586.71	649.21	712.55
		FastBPP	2.76 (120 $\times$)	5.06 (79 $\times$)	9.47 (49 $\times$)	17.91 (29 $\times$)	32.92 (18 $\times$)	62.45 (10 $\times$)	118.95 (6 $\times$)
		FastBPP-V	114.60 (3 $\times$)	117.19 (3 $\times$)	121.45 (4 $\times$)	130.23 (4 $\times$)	144.92 (4 $\times$)	174.89 (4 $\times$)	231.48 (3 $\times$)

A Matrix Size: Dimensions of matrix A.

NRHS: Number of right-hand sides (columns in matrix B).

κ : Condition number of matrix A.

Solve Time: Wall-clock time for complete NNLS solution. Values in parentheses show speedup relative to BPP-BC.

We established our own implementation of the fully GPU-resident BPP solver using batched Cholesky as the baseline because, to our knowledge, no such solver is publicly available. While many NMF frameworks solve multiple right-hand-side NNLS problems, they often use other types of NNLS methods (e.g., Multiplicative Updates [50], HALS [18], or ADMM [108, 93]). Existing GPU-accelerated BPP implementations typically offload only the Cholesky solve to the GPU, leaving other logic on the CPU and incurring significant data transfer overhead [106]. Our baseline ensures a fair comparison, guaranteeing that observed speedups are due to algorithmic improvements, not merely the elimination of CPU-GPU bottlenecks.

Table 6.4. Solver accuracy and BPP iteration count comparison of **FastBPP** variants

A Size	RHS	Solver Type	Relative Error (# BPP iterations)						
			$\kappa=2$	$\kappa=4$	$\kappa=8$	$\kappa=16$	$\kappa=32$	$\kappa=64$	$\kappa=128$
1000×500	25K	BPP-BC	2e-15 (4)	3e-15 (5)	6e-15 (6)	2e-14 (6)	4e-14 (9)	6.4e-11 (11)	3.5e-10 (16)
		FastBPP	9.7e-10 (4)	9.7e-10 (5)	9.5e-10 (6)	9.8e-10 (6)	1.23e-09 (9)	1.87e-09 (11)	3.10e-09 (29)
		FastBPP-V	2e-15 (4)	3e-15 (5)	6e-15 (6)	2e-14 (6)	4e-14 (9)	6.4e-11 (11)	3.5e-10 (29)
	50K	BPP-BC	2e-15 (4)	3e-15 (5)	6e-15 (6)	4.3e-12 (7)	2.2e-11 (7)	1.1e-10 (12)	5.4e-10 (16)
		FastBPP	9.7e-10 (4)	9.7e-10 (5)	9.4e-10 (6)	9.6e-10 (7)	1.22e-09 (7)	1.85e-09 (12)	3.20e-09 (32)
		FastBPP-V	2e-15 (4)	3e-15 (5)	6e-15 (6)	4.3e-12 (7)	2.2e-11 (7)	1.1e-10 (12)	5.4e-10 (32)
	100K	BPP-BC	2e-15 (4)	3e-15 (5)	6e-15 (6)	4.0e-12 (7)	3.2e-11 (8)	1.2e-10 (16)	4.9e-10 (29)
		FastBPP	9.7e-10 (4)	9.7e-10 (5)	9.4e-10 (6)	9.4e-10 (7)	1.20e-09 (8)	1.83e-09 (16)	3.07e-09 (38)
		FastBPP-V	2e-15 (4)	3e-15 (5)	6e-15 (6)	4.0e-12 (7)	3.2e-11 (8)	1.1e-10 (16)	4.8e-10 (38)
2000×1000	25K	BPP-BC	2e-15 (4)	3e-15 (5)	7e-15 (6)	4.3e-12 (7)	1.2e-11 (7)	6.1e-11 (11)	6.1e-10 (17)
		FastBPP	6.7e-10 (4)	7.1e-10 (5)	7.3e-10 (6)	7.4e-10 (7)	9.2e-10 (7)	1.41e-09 (11)	2.61e-09 (17)
		FastBPP-V	2e-15 (4)	3e-15 (5)	7e-15 (6)	4.3e-12 (7)	1.2e-11 (7)	6.1e-11 (11)	6.1e-10 (17)
	50K	BPP-BC	2e-15 (4)	3e-15 (5)	2.0e-12 (6)	4.9e-12 (7)	3.4e-11 (8)	1.5e-10 (13)	5.9e-10 (23)
		FastBPP	6.7e-10 (4)	7.1e-10 (5)	7.3e-10 (6)	7.3e-10 (7)	9.2e-10 (8)	1.44e-09 (13)	2.54e-09 (23)
		FastBPP-V	2e-15 (4)	3e-15 (5)	1.8e-12 (6)	4.9e-12 (7)	3.4e-11 (8)	1.5e-10 (13)	5.9e-10 (23)
	100K	BPP-BC	9e-14 (5)	3e-13 (5)	1.6e-12 (6)	7.4e-12 (7)	2.4e-11 (10)	1.3e-10 (16)	5.1e-10 (35)
		FastBPP	6.7e-10 (5)	7.1e-10 (5)	7.2e-10 (6)	7.1e-10 (7)	9.0e-10 (10)	1.39e-09 (16)	2.40e-09 (35)
		FastBPP-V	9e-14 (5)	3e-13 (5)	1.6e-12 (6)	7.4e-12 (7)	2.1e-11 (10)	1.3e-10 (16)	5.0e-10 (35)
4000×2000	25K	BPP-BC	3e-15 (4)	5e-15 (5)	9e-13 (6)	7.1e-12 (7)	3.5e-11 (7)	1.4e-10 (10)	5.9e-10 (21)
		FastBPP	4.6e-10 (4)	5.1e-10 (5)	5.0e-10 (6)	5.3e-10 (7)	7.0e-10 (7)	1.08e-09 (10)	2.01e-09 (21)
		FastBPP-V	3e-15 (4)	5e-15 (5)	9e-13 (6)	7.1e-12 (7)	3.5e-11 (7)	1.4e-10 (10)	5.9e-10 (21)
	50K	BPP-BC	2e-13 (5)	1e-13 (5)	1.5e-12 (6)	1.1e-11 (7)	4.8e-11 (8)	1.5e-10 (16)	6.1e-10 (21)
		FastBPP	4.6e-10 (5)	5.1e-10 (5)	5.0e-10 (6)	5.3e-10 (7)	6.7e-10 (8)	1.06e-09 (17)	2.02e-09 (24)
		FastBPP-V	2e-13 (5)	1e-13 (5)	1.5e-12 (6)	1.1e-11 (7)	4.8e-11 (8)	1.5e-10 (17)	6.1e-10 (24)
	100K	BPP-BC	3e-13 (5)	9e-13 (5)	4.1e-12 (6)	9.6e-12 (7)	3.8e-11 (8)	1.2e-10 (17)	5.7e-10 (44)
		FastBPP	4.5e-10 (5)	4.9e-10 (5)	5.0e-10 (6)	5.1e-10 (7)	6.3e-10 (8)	9.5e-10 (18)	1.94e-09 (46)
		FastBPP-V	3e-13 (5)	9e-13 (5)	4.1e-12 (6)	9.6e-12 (7)	3.8e-11 (8)	1.2e-10 (18)	5.5e-10 (46)

A Matrix Size: Dimensions of matrix A.

NRHS: Number of right-hand sides (columns in matrix B).

κ : Condition number of matrix A.

Relative Error: Solution accuracy relative to ground truth. Values in parentheses show NNLS iterations.

6.4.6.3 Performance

The results in Table 6.9 and Table 6.3 demonstrate significant computational advantages of the **FastBPP** variants over the baseline approach. **FastBPP** achieves substantial speedups, particularly for well-conditioned systems ($\kappa = 2 - 16$), with performance gains up to $120.2\times$ faster than **BPP-BC**. **FastBPP-V** exhibits more modest but still notable speedups of upto $4.1\times$, reflecting the computational overhead of the verification step while maintaining superior performance compared to the baseline. As expected, the performance advantage diminishes for ill-conditioned systems ($\kappa \geq 64$) due to the increased conjugate gradient iterations required for convergence, highlighting a potential research direction for developing more so-

phisticated column grouping strategies and targeted preconditioning techniques for high-condition-number systems.

6.4.6.4 Solution Accuracy Assessment

All solver variants achieve excellent numerical accuracy, with relative errors ranging from 10^{-15} to 10^{-9} as shown in Table 6.4. Notably, **FastBPP-V**'s solution quality is virtually identical to the baseline, confirming that the direct solver verification step successfully maintains numerical exactness. Even without this check, the standard **FastBPP** maintains very low relative errors, demonstrating that its adaptive iterative solver preserves numerical stability while providing significant performance gains.

Beyond the final solution quality, the number of BPP iterations—the steps needed to find the optimal active set—reveals how each solver behaves. For well-conditioned problems ($\kappa \leq 64$), the BPP iteration counts are consistent across all methods. This indicates that the approximate solutions from the Block Masked CG solver are precise enough to effectively guide the active-set algorithm along the correct path. A divergence appears only in the ill-conditioned case ($\kappa = 128$), where **FastBPP** based solvers occasionally requires more BPP iterations than the baseline. This occurs because the reduced accuracy of the iterative solver on these ill-conditioned problems can temporarily misguide the partitioning step. This temporary misguidance is ultimately inconsequential, as the BPP method's inherent robustness in finding the optimal passive set ensures the FastBPP solvers still converge to the correct optimal solution.

6.5 Conclusion

In this chapter, we address our fourth research question: **RQ4**: Can we accelerate the constrained optimization problem in constrained sparse tensor factorization by exploiting the inherent computational structure? We make the following contributions:

1. We present a *Blocked & Fused* ADMM algorithm optimized for multicore shared memory systems. By employing data blocking to maximize parallelism and operation fusion to eliminate redundant loads and stores through register reuse, we reduce the algorithm’s memory traffic by over 30%. This alleviates the memory bandwidth bottleneck inherent to ADMM, achieving speedups of up to $8.1\times$ compared to a standard parallel baseline on a 56-core CPU.
2. We introduce *FastBPP*, a high-throughput, GPU-resident non-negative least squares solver based on the Block Principal Pivoting method. We propose a novel *Block Masked Preconditioned Conjugate Gradient* (PBCG) solver that utilizes the global system matrix’s Cholesky factor as a preconditioner to accelerate convergence for dense passive sets. By dynamically switching between PBCG and standard Block CG based on column sparsity, and employing a selective direct-solver verification step, FastBPP achieves up to $120.2\times$ speedup over state-of-the-art batched Cholesky approaches on an NVIDIA H100 GPU while maintaining numerical accuracy.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

The overarching goal of this dissertation is to address the multiple challenges in computing sparse tensor factorization in an *efficient* and *scalable* manner.

We lay out the following fundamental challenges in Chapter 1:

- The *extreme sparse* nature of real-world streaming tensors hinder the scalability of sparse tensor factorization algorithms.
- Applications of constraints such as *non-negativity* expose additional computational challenges in terms of computational efficiency and parallelization

Throughout the course of this dissertation, we have addressed these challenges through: novel algorithmic innovations, adaptive sparse tensor data structures, and hardware-specific optimizations for modern parallel computing architectures. We have advanced the state-of-the-art in sparse tensor factorization to enable efficient processing of large-scale sparse and streaming tensors while maintaining the interpretability and physical meaning required by real-world applications through constrained factorization.

7.2 Key Contributions by Research Questions

In this section, we summarize the key contributions of this dissertation by answering the research questions outlined in Section 1.1.

RQ1: How can we optimize streaming sparse tensor factorization throughput by exploiting the hyper-sparse nature of the streaming tensors? We have proposed a novel algorithm, Row-Sparse CP-stream, for stream-

ing sparse tensor factorization that exploits the hyper-sparse nature of streaming tensors. By partitioning the factor matrices into two non-overlapping subsets of zero-rows and non-zero-rows and updating them independently, we reduce the required memory footprint for sparse MTTKRP operations. Furthermore, by storing and computing these subsets in Gram matrix form, we greatly reduce the required arithmetic operations, resulting in a high-performance streaming CP decomposition algorithm. Our new algorithm achieves up to $102\times$ speedup over the original non-constrained CP-stream algorithm.

RQ2: Can we further improve the throughput of streaming sparse tensor factorization utilizing adaptive linearized sparse tensor formats?

We have developed ALTO-stream, a novel framework that significantly improves streaming sparse tensor factorization throughput through adaptive linearized sparse tensor formats. Our approach introduces a *dynamic tensor linearization* scheme that quickly generates compressed tensor slices on-the-fly and selects the best nonzero element traversal and synchronization strategy based on the characteristics of each slice. This is accomplished by using a *spatial sketching* algorithm that retains all incoming nonzero elements at a given time instance but remaps them into a tensor sketch with reduced multi-dimensional image space. The coordinates are then linearized into a single index and the nonzero elements are sorted accordingly. This compact format greatly reduces memory usage and improves the data locality of streaming tensor factorization with minimal format generation overhead (Sec. 4.3.1 and 4.3.2).

To further enhance throughput, we introduce *dynamic time slicing* (DTS), a tunable mechanism that balances the frequency of factor updates with efficient utilization of computing resources. DTS uses variable-width time slices (instead of a fixed width) by aggregating interactions across time slices until some minimum slice

weight (*i.e.*, number of nonzero elements) is achieved or a user-provided update frequency is reached. This improves the overall resource utilization and increases the achieved performance while maintaining the accuracy of the original algorithm (Sec. 4.3.3).

We demonstrate the efficacy of ALTO-stream by accelerating the state-of-the-art CP-stream [86] and spCP-stream algorithms [91], improving their performance by $10.3\text{--}11\times$ and $6.4\text{--}7.2\times$, respectively, for a range of real-world streaming tensors on a 56-core CPU.

RQ3: Can we develop GPU-optimized constrained update algorithms to enable efficient end-to-end GPU-resident sparse tensor factorization for real-world applications?

We have developed the first fully GPU-resident framework for end-to-end constrained sparse tensor factorization (cSTF), addressing the fundamental shift in performance bottlenecks for sparse tensors. Unlike dense tensor factorization where MTTKRP dominates computation time, sparse tensor factorization is bottlenecked by constrained update methods. Our framework integrates the alternating optimization with alternating direction method of multipliers (AO-ADMM), a fast and robust constraint enforcement algorithm, to fully operate on the GPU. We leverage state-of-the-art GPU sparse tensor formats (BLCO [67]) and perform all computations—from MTTKRP to factor matrix updates—entirely on the GPU, eliminating costly CPU-GPU data transfers.

Through systematic performance analysis of the ADMM algorithm on GPUs, we identified critical bottlenecks related to redundant computation and data movement. We propose cuADMM, an optimized ADMM implementation that applies operation fusion and pre-inversion techniques to eliminate these inefficiencies. cuADMM achieves a $1.8\times$ geometric mean speedup over generic ADMM implementations

across 10 real-world sparse tensors on an NVIDIA H100 GPU. By fully offloading cSTF to the GPU, our framework outperforms the state-of-the-art CPU-based library SPLATT [87], achieving geometric mean speedups of $5.1\times$ on NVIDIA A100 and $7.01\times$ on H100 GPUs, with maximum speedups of $41.59\times$ and $58.05\times$, respectively.

We further demonstrate the flexibility and generalizability of our framework by incorporating additional GPU-based non-negativity constraint algorithms, including HALS and MU, as well as the state-of-the-art sparse MTTKRP GPU kernel [67]. These implementations achieve significant speedups over their CPU counterparts, validating that our GPU-resident framework provides a robust foundation for various constrained tensor factorization methods.

RQ4: Can we accelerate the constrained optimization problem in factorization algorithms by exploiting the inherent computational structure? We focus on developing scalable implementations of two representative NNLS solvers, ADMM and BPP, across different computing architectures. For ADMM on multicore shared memory systems, we identify that the algorithm is highly memory bandwidth-bound due to low arithmetic intensity operations. We propose Blocked & Fused ADMM, which applies data blocking to improve cache locality, operation fusion to eliminate intermediate data movement, and parallel reduction for column-wise operations. These optimizations reduce memory traffic by over 30% and achieve up to $8.1\times$ speedup with 56 threads on real-world sparse tensors.

For active-set based methods, we address the fundamental parallelization challenge in multiple right-hand side (RHS) NNLS problems: each column requires solving a structurally diverse linear system with a unique active set, preventing efficient batching on GPUs. The Block Principal Pivoting (BPP) algorithm identifies all KKT-violating variables and performs block pivots to accelerate conver-

gence, but the mask-constrained normal equation solve step consumes the majority of computational time. Direct solvers using batched Cholesky factorization suffer from $O(n_p^3)$ complexity per column, memory fragmentation from irregular gather-scatter operations, and load imbalance. Traditional iterative Conjugate Gradient methods, while asymptotically faster at $O(n_{cg}n_p^2)$, face poor GPU utilization due to Level-1/2 BLAS operations and cannot exploit block methods since each column requires a different system matrix based on its passive set.

We propose FastBPP, a GPU-accelerated framework featuring Block Masked Preconditioned CG (PBCG) that uses the pre-computed Cholesky factorization of the global Gram matrix ($\mathbf{A}^T \mathbf{A}$) as a preconditioner for column-specific subproblems. Through convergence analysis via the Woodbury matrix identity, we prove that when k variables are in the active set, the preconditioned system becomes identity plus a rank- k matrix, guaranteeing convergence in at most $k + 1$ iterations. Our adaptive strategy dynamically switches between PBCG for dense passive set columns and Block Masked CG for sparse ones, maximizing efficiency across varying sparsity patterns. To maintain numerical exactness, we introduce selective verification where recently converged columns are verified using direct solvers. FastBPP achieves up to $120.2\times$ speedup over batched Cholesky baselines on well-conditioned systems ($\kappa \leq 16$) and maintains excellent numerical accuracy with relative errors ranging from 10^{-15} to 10^{-9} across 945 synthetic test problems on an NVIDIA H100 GPU, demonstrating that iterative exploration with direct verification provides an optimal balance between computational efficiency and solution accuracy.

7.3 Future Research Directions

The research contributions presented in this dissertation have laid the foundation for future work in scaling constrained factorization algorithms for real-world applications and exploring data-driven space filling curves for sparse tensor formats.

In this section, we outline promising future research directions that are currently being actively pursued.

7.3.1 Scaling up Constrained Factorization Algorithms for Domain Science Applications

We are extending the scalable NNLS solvers developed in this dissertation to two diverse domains: computational materials science and computational biology.

In the first domain, we apply non-negativity constrained factorization to study the process of recrystallization in molybdenum disulfide (MoS_2). Non-negativity constrained factorization allows us to *reduce* the complexity of analyzing millions of atomic configurations across time frames—each with dozens of features per atom—by identifying recurring structural motifs and tracking their evolution, offering insight into the kinetics and thermodynamics of material phase transitions (See Figure 7.1). This application is particularly important because MoS_2 , a two-dimensional (2D) material with various electrical, optical, and mechanical properties, is of growing interest for next-generation transistors, batteries, and flexible electronics [102, 38].

However, the performance of MoS_2 -based devices is highly sensitive to atomic-scale defects and grain boundaries that emerge during synthesis and thermal treatment, making the recrystallization process—by which disordered or amorphous regions transform into more ordered crystalline structures—a key factor in tuning

device properties. Extracting latent structural features from the 2D lattice enables characterization of these defects and boundaries at scale, providing insights into how they evolve during recrystallization.

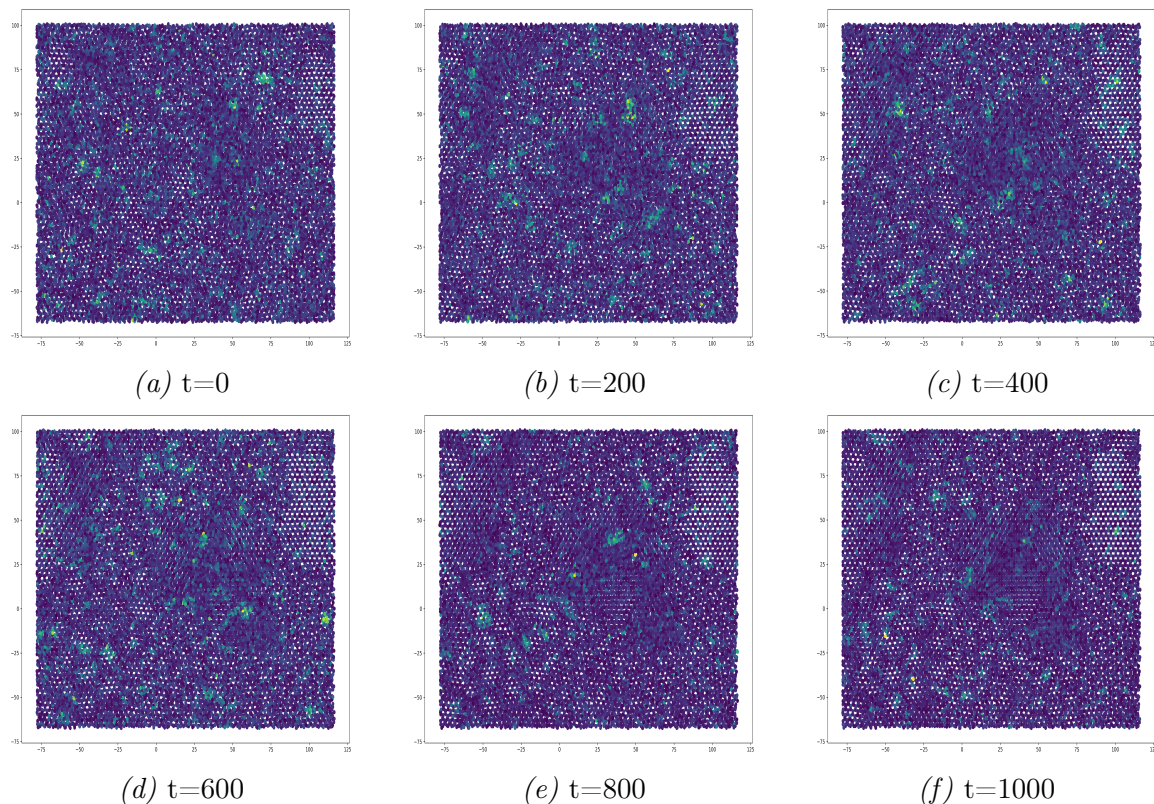


Figure 7.1. Spatiotemporal evolution of structural phase transitions in a 2D material lattice, derived from NMF decomposition of molecular dynamics simulation (MDS) data, projected onto atomic positions at selected timesteps ($t = 0, 200, 400, 600, 800, 1000$). Lightly colored regions indicate the regions of the atomic lattice that are undergoing structural phase transitions.

In the second domain, we apply NMF as a biologically interpretable dimensionality reduction technique for clustering protein families from large-scale biological sequence data. NMF discovers latent factors—analogueous to biochemical or evolutionary themes—that group functionally similar proteins, mirroring topic modeling in natural language processing where documents are grouped based on shared topics inferred from word distributions [51] or clustering multi-dimensional data

[36, 58, 81, 94, 4, 107, 5, 62]. This application addresses the computational challenge of analyzing massive datasets: protein sequences encoded as high-dimensional k-mer representations (where each dimension corresponds to the frequency of a short substring) can result in matrices with hundreds of thousands of possible k-mers and millions of protein sequences, containing tens of billions of entries. Scalable NMF algorithms enabled by efficient NNLS solvers are essential for understanding protein function, predicting evolutionary relationships, and annotating newly sequenced genomes at this scale.

7.3.2 Data-Driven Space Filling Curves for Sparse Tensors

Sparse tensor operations on real-world tensors are inherently data-dependent as their workload is determined by a granular level of non-zero elements or non-zero fibers of smaller subblocks. As such, deriving optimized sparse tensor formats is a challenging problem and state-of-the-art approaches often rely on suboptimal heuristics that fail to account for the irregular shapes and/or data distributions of sparse tensors.

In Chapter 4, we introduced the streaming adaptation of the adaptive linearized tensor order (ALTO) framework [32], which significantly improved sparse tensor operation performance by tailoring tensor representations to the irregular shapes of streaming sparse tensors through dynamic linearization and spatial sketching. ALTO-based formats demonstrated that adapting data layout to tensor geometry can dramatically enhance data locality and computational efficiency. However, while these formats adapt to the asymmetric shapes of sparse tensors, they rely on heuristic-based space-filling curves that do not fully exploit the underlying data distributions within tensors. This observation motivates an important

future research direction: can we leverage data-driven methodologies to automatically discover even more efficient sparse tensor encodings that adapt to both tensor geometry *and* data distribution patterns?

Building upon the ALTO framework, we are pursuing the reinforcement-learned adaptive tensor encoding (RELATE) framework. RELATE addresses a fundamental challenge in sparse tensor optimization: the optimization space for tensor decomposition problems is vast due to their high-dimensional and data-dependent nature, yet existing approaches—including ALTO—rely on fixed heuristics that cannot adapt to the full spectrum of irregular data distributions encountered in real-world tensors. Prior attempts to use supervised machine learning for format selection [95, 96] are limited by the lack of large-scale labeled training sets and the absence of known optimal solutions (ground truth) for such irregular workloads. Moreover, these approaches are fundamentally bounded by the performance of existing formats, even with oracle-level prediction accuracy.

RELATE takes a fundamentally different approach by adopting the emerging learned algorithm paradigm [65]. Rather than selecting among predefined formats or following fixed heuristics, RELATE employs deep reinforcement learning (DRL) to *construct* efficient sparse tensor encodings along with corresponding schedules of tensor compute and memory operations—all without requiring labeled training data. Central to RELATE is an autonomous agent that learns optimal sparse tensor representations through interactions with the tensor decomposition environment, observing state and reward signals to discover encodings that adapt to both tensor geometry and data distributions.

The key innovation in RELATE is a hybrid model-free and model-based learning algorithm that navigates the vast solution space through several mechanisms:

- An adaptive neural network architecture that scales with environment complexity and learns from both evaluated (real) and estimated (imagined) actions to handle large-scale tensors effectively.
- *Rule-driven action masking* to prune functionally invalid actions during early exploration, ensuring that RELATE always generates correct sparse representations even before fully learning environment dynamics.
- *Dynamics-informed action filtering* to eliminate low-value actions during exploitation, accelerating convergence to high-quality solutions.
- A decoupled server-client execution model with relative reward signals to ensure accurate performance measurements in noisy HPC environments [98].

By overcoming the limitations of both expert-designed heuristics (including ALTO) and supervised learning methods, RELATE represents a promising direction for automatically adapting sparse tensor formats to the full complexity of real-world data. Preliminary results indicate that learned encodings can outperform expert-designed formats, with geometric-mean speedups of $1.4\times$ – $1.46\times$ over the best existing formats, and up to $2\times$ speedup for large-scale, low-density tensors that are particularly challenging to optimize. This future direction has the potential to establish a new paradigm for sparse tensor format design, where machine learning discovers representations that go beyond what domain experts can manually devise.

BIBLIOGRAPHY

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: A System for Large-Scale Machine Learning*, Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, USENIX Association, Savannah, GA, USA, 2016, 265–283, event-place: Savannah, GA, USA.
- [2] Harbir Antil, Sergey Dolgov, and Akwum Onwunta, *TTRISK: Tensor train decomposition algorithm for risk averse optimization*, Numer. Linear Algebra Appl. **30** (2023), no. 3, e2481, DOI: <https://doi.org/10.1002/nla.2481>, eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/nla.2481>.
- [3] Brett W. Bader and Tamara G. Kolda, *Efficient MATLAB computations with sparse and factored tensors*, SIAM JOURNAL ON SCIENTIFIC COMPUTING **30** (2007), no. 1, 205–231, DOI: 10.1137/060676489.
- [4] Arindam Banerjee, Inderjit Dhillon, Joydeep Ghosh, Srujana Merugu, and Dharmendra S. Modha, *A Generalized Maximum Entropy Approach to Bregman Co-Clustering and Matrix Approximation*, Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '04, Association for Computing Machinery, Seattle, WA, USA, 2004, 509–514, DOI: 10.1145/1014052.1014111.
- [5] Arindam Banerjee, Chase Krumpelman, Joydeep Ghosh, Sugato Basu, and Raymond J. Mooney, *Model-Based Overlapping Clustering*, Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD '05, Association for Computing Machinery, Chicago, Illinois, USA, 2005, 532–537, DOI: 10.1145/1081870.1081932.
- [6] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin, *Efficient and Scalable Computations with Sparse Tensors*, 2012 IEEE Conference on High Performance Extreme Computing, 2012, 1–6, DOI: 10.1109/HPEC.2012.6408676, DOI: 10.1109/HPEC.2012.6408676.
- [7] Åke Björck, *Numerical Methods for Least Squares Problems*, Society for Industrial and Applied Mathematics, 1996, DOI: 10.1137/1.9781611971484, eprint: <https://epubs.siam.org/doi/pdf/10.1137/1.9781611971484>.

- [8] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra, *Matrix Market: a web resource for test matrix collections*, Quality of Numerical Software: Assessment and enhancement, Springer US, Boston, MA, 1997, 125–137, DOI: 10.1007/978-1-5041-2940-4_9.
- [9] A. W. Bojanczyk, R. P. Brent, P. van Dooren, and F. R. de Hoog, *A Note on DOWDATING the Cholesky Factorization*, SIAM J. Sci. Stat. Comput. **8** (1987), no. 3, 210–221.
- [10] Rasmus Bro and Sijmen Jong, *A Fast Non-negativity-constrained Least Squares Algorithm* (en), Journal of Chemometrics **11** (1997), no. 5, 393–401, DOI: 10.1002/(SICI)1099-128X(199709/10)11:53.0.CO;2-L.
- [11] David Bruns-Smith, Muthu M. Baskaran, James Ezick, Tom Henretty, and Richard Lethin, *Cyber Security through Multidimensional Data Decompositions*, 2016 Cybersecurity Symposium (CYBERSEC), 2016, 59–67, DOI: 10.1109/CYBERSEC.2016.017.
- [12] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson, *Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks*, Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '09, Association for Computing Machinery, Calgary, AB, Canada, 2009, 233–244, DOI: 10.1145/1583991.1584053, event-place: Calgary, AB, Canada.
- [13] Jason Cantarella and Michael Piatek, *tsnnls: A solver for large sparse least squares problems with non-negative variables* (en) ().
- [14] J Douglas Carroll and Jih-Jie Chang, *Analysis of individual differences in multidimensional scaling via an N-way generalization of “Eckart-Young” decomposition*, Psychometrika, Vol. 35, **3**, Springer, 1970, 283–319.
- [15] Venkatesan T. Chakaravarthy, Jee W. Choi, Douglas J. Joseph, Prakash Murali, Shivmaran S. Pandian, Yogish Sabharwal, and Dheeraj Sreedhar, *On Optimizing Distributed Tucker Decomposition for Sparse Tensors*, Proceedings of the 2018 International Conference on Supercomputing, ICS '18, Association for Computing Machinery, Beijing, China, 2018, 374–384, DOI: 10.1145/3205289.3205315.
- [16] Jee Choi, Xing Liu, Shaden Smith, and Tyler Simon, *Blocking Optimization Techniques for Sparse Tensor Computation*, 2018 IEEE International Paral-

- lel and Distributed Processing Symposium (IPDPS), IEEE, USA, 2018, 568–577, DOI: 10.1109/IPDPS.2018.00066, DOI: 10.1109/IPDPS.2018.00066.
- [17] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe, *Format Abstraction for Sparse Tensor Algebra Compilers*, Proc. ACM Program. Lang. **2** (2018), no. OOPSLA, DOI: 10.1145/3276493, Place: New York, NY, USA Publisher: Association for Computing Machinery.
 - [18] Andrzej Cichocki and Anh-Huy Phan, *Fast Local Algorithms for Large Scale Nonnegative Matrix and Tensor Factorizations* (en), IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **E92-A** (2009), no. 3, 708–721, DOI: 10.1587/transfun.E92.A.708.
 - [19] Gautam Dasarathy, Parikshit Shah, Badri Narayan Bhaskar, and Robert D Nowak, *Sketching Sparse Matrices, Covariances, and Graphs Via Tensor Products*, IEEE Transactions on Information Theory **61** (2015), no. 3, 1373–1388.
 - [20] Monica Dessoie, Marco Dell’Orto, and Fabio Marcuzzi, *The Lawson-Hanson algorithm with deviation maximization: Finite convergence and sparse recovery*, Numer. Linear Algebra Appl. **30** (2023), no. 5, DOI: 10.1002/nla.2490.
 - [21] Jelena Diakonikolas, Chenghui Li, Swati Padmanabhan, and Chaobing Song, *A fast scale-invariant algorithm for non-negative least squares with non-negative data*, Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS ’22, Curran Associates Inc., New Orleans, LA, USA, 2022.
 - [22] S. V. Dolgov, B. N. Khoromskij, and I. V. Oseledets, *Fast Solution of Parabolic Problems in the Tensor Train/Quantized Tensor Train Format with Initial Application to the Fokker–Planck Equation*, SIAM J. Sci. Comput. **34** (2012), no. 6, A3016–A3038, DOI: 10.1137/120864210.
 - [23] Srinivas Eswar, Koby Hayashi, Grey Ballard, Ramakrishnan Kannan, Michael A. Matheson, and Haesun Park, *PLANC: Parallel Low-rank Approximation with Nonnegativity Constraints* (en), ACM Transactions on Mathematical Software **47** (2021), no. 3, 1–37, DOI: 10.1145/3432185.
 - [24] Hadi Fanaee-T and João Gama, *Tensor-based anomaly detection: An interdisciplinary survey*, Knowl. Based Syst. **98** (2016), 130–147, DOI: 10.1016/j.knosys.2016.01.027.

- [25] Sofia Fernandes, Hadi Fanaee-T, and João Gama, *Evolving Social Networks Analysis via Tensor Decompositions: From Global Event Detection Towards Local Pattern Discovery and Specification*, Discovery Science, Springer International Publishing, Cham, 2019, 385–395.
- [26] Gene H. Golub and Charles F. Van Loan, *Matrix Computations*, Third, The Johns Hopkins University Press, 1996.
- [27] Lars Grasedyck, Melanie Kluge, and Sebastian Krämer, *Variants of Alternating Least Squares Tensor Completion in the Tensor Train Format*, SIAM J. Sci. Comput. **37** (2015), no. 5, A2424–A2450, DOI: 10.1137/130942401.
- [28] Anne Greenbaum, *Iterative Methods for Solving Linear Systems, Vol. 17*, Frontiers in Applied Mathematics, SIAM, 1997.
- [29] Adam Harrison and Dileepan Joseph, *High Performance Rearrangement and Multiplication Routines for Sparse Tensor Arithmetic*, SIAM J. Sci. Comput. **40** (2018), no. 2, C258–C281, DOI: 10.1137/17M1115873, DOI: 10.1137/17M1115873.
- [30] Richard A Harshman, *Foundations of the parafac procedure: Models and conditions for an "explanatory" multimodal factor analysis*, UCLA working papers in phonetics **16** (1970), no. 1, 1–84.
- [31] Huan He, Jette Henderson, and Joyce C Ho, *Distributed Tensor Decomposition for Large Scale Health Analytics*, The World Wide Web Conference, WWW '19, Association for Computing Machinery, San Francisco, CA, USA, 2019, 659–669, DOI: 10.1145/3308558.3313548, event-place: San Francisco, CA, USA.
- [32] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi, *ALTO: Adaptive Linearized Storage of Sparse Tensors* (en), Proceedings of the ACM International Conference on Supercomputing, ICS '21, Association for Computing Machinery, Virtual Event, USA, 2021, 404–416, DOI: 10.1145/3447818.3461703, arXiv:2102.10245 [cs].
- [33] Joyce C. Ho, Joydeep Ghosh, and Jimeng Sun, *Marble: High-Throughput Phenotyping from Electronic Health Records via Sparse Nonnegative Tensor Factorization*, Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, Association

- for Computing Machinery, New York, New York, USA, 2014, 115–124, DOI: 10.1145/2623330.2623658, event-place: New York, New York, USA.
- [34] Anh Huy Phan and Andrzej Cichocki, *PARAFAC algorithms for large-scale problems*, Neurocomputing **74** (2011), no. 11, 1970–1984, DOI: <https://doi.org/10.1016/j.neucom.2010.06.030>, Adaptive Incremental Learning in Neural Networks Learning Algorithm and Mathematic Modelling Selected papers from the International Conference on Neural Information Processing 2009 (ICONIP 2009).
 - [35] Intel Corporation, *Intel® VTune™ Profiler*, URL: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.htm>, 2022.
 - [36] Stefanie Jegelka, Suvrit Sra, and Arindam Banerjee, *Approximation Algorithms for Tensor Clustering*, International Conference on Algorithmic Learning Theory, Springer, 2009, 368–383.
 - [37] Joaquim J. Júdice and Fernanda M. Pires, *A block principal pivoting algorithm for large-scale strictly monotone linear complementarity problems*, Comput. Oper. Res. **21** (1994), no. 5, 587–596, DOI: [https://doi.org/10.1016/0305-0548\(94\)90106-6](https://doi.org/10.1016/0305-0548(94)90106-6).
 - [38] Sergei V Kalinin, Mark P Oxley, Mani Valleti, Junjie Zhang, Raphael P Hermann, Hong Zheng, Wenrui Zhang, Gyula Eres, Rama K Vasudevan, and Maxim Ziatdinov, *Deep Bayesian local crystallography*, npj Computational Materials **7** (2021), no. 1, 181.
 - [39] Oguz Kaya, *High Performance Parallel Algorithms for Tensor Decompositions*, Theses, Université de Lyon, 2017.
 - [40] Oguz Kaya and Bora Uçar, *High-performance parallel algorithms for the Tucker decomposition of higher order sparse tensors* (2015).
 - [41] Dongmin Kim, Suvrit Sra, and Inderjit S. Dhillon, *Fast Newton-type Methods for the Least Squares Nonnegative Matrix Approximation Problem*, Proceedings of the 2007 SIAM International Conference on Data Mining (SDM), 343–354, DOI: 10.1137/1.9781611972771.31.
 - [42] Jingu Kim and Haesun Park, *Fast Nonnegative Matrix Factorization: An Active-Set-Like Method and Comparisons* (en), SIAM J. Sci. Comput. **33** (2011), no. 6, 3261–3281, DOI: 10.1137/110821172.

- [43] David B Kirk, *NVIDIA CUDA Software and GPU Parallel Computing Architecture* (2006).
- [44] Teruyoshi Kobayashi, Anna Sapienza, and Emilio Ferrara, *Extracting the multi-timescale activity patterns of online financial markets*, Scientific Reports **8** (2018), no. 1, 1–11, Publisher: Nature Publishing Group.
- [45] Milad Kokhazadeh, Georgios Keramidas, Vasilios Kelefouras, and Iakovos Stamoulis, *A Practical Approach for Employing Tensor Train Decomposition in Edge Devices*, Int. J. Parallel Program. **52** (2024), no. 1–2, 20–39, DOI: 10.1007/s10766-024-00762-3.
- [46] Tamara G. Kolda and Brett W. Bader, *Tensor Decompositions and Applications* (en), SIAM Rev. **51** (2009), no. 3, 455–500, DOI: 10.1137/07070111X, DOI: 10.1137/07070111X.
- [47] Tamara G. Kolda and Jimeng Sun, *Scalable Tensor Decompositions for Multi-aspect Data Mining*, 2008 Eighth IEEE International Conference on Data Mining, ICDM '08, IEEE Computer Society, USA, 2008, 363–372, DOI: 10.1109/ICDM.2008.89, DOI: 10.1109/ICDM.2008.89.
- [48] Jan Laukemann, Ahmed E. Helal, S. Isaac Geronimo Anderson, Fabio Checconi, Yongseok Soh, Jesmin Jahan Tithi, Teresa Ranadive, Brian J. Gravelle, Fabrizio Petrini, and Jee Choi, *Accelerating Sparse Tensor Decomposition Using Adaptive Linearized Representation*, IEEE Transactions on Parallel and Distributed Systems **36** (2025), no. 5, 1025–1041, DOI: 10.1109/TPDS.2025.3553092, DOI: 10.1109/TPDS.2025.3553092.
- [49] Charles Lawson and Richard Hanson, *Solving Least Squares Problems / SIAM Publications Library* (en), 1995.
- [50] Daniel D. Lee and H. Sebastian Seung, *Learning the parts of objects by non-negative matrix factorization* (en), Nature **401** (1999), no. 6755, 788–791, DOI: 10.1038/44565.
- [51] Daniel D. Lee and H. Sebastian Seung, *Learning the parts of objects by non-negative matrix factorization*, Nature **401** (1999), 788–791.
- [52] J. Li, J. Sun, and R. Vuduc, *HiCOO: Hierarchical Storage of Sparse Tensors* (en), SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, SC '18, IEEE, Dallas, Texas, 2018, 238–252, DOI: 10.1109/SC.2018.00022, DOI: 10.1109/SC.2018.00022.

- [53] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc, *Efficient and Effective Sparse Tensor Reordering* (en), Proceedings of the ACM International Conference on Supercomputing, ICS '19, Association for Computing Machinery, Phoenix, Arizona, 2019, 227–237, DOI: 10.1145/3330345.3330366, event-place: Phoenix, Arizona.
- [54] Lingjie Li, Wenjian Yu, and Kim Batselier, *Faster tensor train decomposition for sparse data*, J. Comput. Appl. Math. **405** (2022), no. C, DOI: 10.1016/j.cam.2021.113972.
- [55] Athanasios P. Liavas, Georgios Kostoulas, Georgios Lourakis, Kejun Huang, and Nicholas D. Sidiropoulos, *Nesterov-Based Alternating Optimization for Nonnegative Tensor Factorization: Algorithm and Parallel Implementation* (en), IEEE Trans. Signal Process. **66** (2018), no. 4, 944–953, DOI: 10.1109/TSP.2017.2777399.
- [56] Chih-Jen Lin, *Projected Gradient Methods for Nonnegative Matrix Factorization*, Neural Comput. **19** (2007), no. 10, 2756–2779, DOI: 10.1162/neco.2007.19.10.2756.
- [57] Bangtian Liu, Chengyao Wen, Anand D. Sarwate, and Maryam Mehri Dehnavi, *A Unified Optimization Approach for Sparse Tensor Operations on GPUs*, 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017, 47–57, DOI: 10.1109/CLUSTER.2017.75, DOI: 10.1109/CLUSTER.2017.75.
- [58] Chaoyue Liu and Mikhail Belkin, *Clustering with Bregman Divergences: an Asymptotic Analysis*, Advances in Neural Information Processing Systems, Vol. 29, Curran Associates, Inc., 2016.
- [59] Shangzhi Liu and Götz Trenkler, *Hadamard, Khatri-Rao, Kronecker, and Other Matrix Products*, International Journal of Information and Systems Sciences **4** (2008), no. 1, 160–177, DOI: 10.1155/2016/8301709.
- [60] Xingyi Liu and Keshab K. Parhi, *Tensor Decomposition for Model Reduction in Neural Networks: A Review [Feature]*, IEEE Circuits and Systems Magazine **23** (2023), no. 2, 8–28, DOI: 10.1109/MCAS.2023.3267921.
- [61] Yuancheng Luo and Ramani Duraiswami, *Efficient Parallel Nonnegative Least Squares on Multicore Architectures*, SIAM J. Sci. Comput. **33** (2011), no. 5, 2848–2863, DOI: 10.1137/100799083, Publisher: Society for Industrial and Applied Mathematics.

- [62] S.C. Madeira and A.L. Oliveira, *Biclustering algorithms for biological data analysis: a survey*, IEEE/ACM Transactions on Computational Biology and Bioinformatics **1** (2004), no. 1, 24–45, DOI: 10.1109/TCBB.2004.2.
- [63] Maryam Mehri Dehnavi, Yaser El-Kurdi, James Demmel, and Dennis Ghodsi, *A Unified Optimization Approach for Sparse Tensor Operations on GPUs*, 2017 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2017, 47–57, DOI: 10.1109/CLUSTER.2017.68.
- [64] M. Merritt and Yin Zhang, *Interior-Point Gradient Method for Large-Scale Totally Nonnegative Least Squares Problems*, J. Optim. Theory Appl. **126** (2005), no. 1, 191–202, DOI: 10.1007/s10957-005-2668-z.
- [65] Michael Mitzenmacher and Sergei Vassilvitskii, *Algorithms with Predictions*, Communications of the ACM **65** (2022), no. 7, 33–35, DOI: 10.1145/3528087.
- [66] J. M. Myre, E. Frahm, D. J. Lilja, and M. O. Saar, *TNT-NN: A Fast Active Set Method for Solving Large Non-Negative Least Squares Problems* (en), Procedia Computer Science, (International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland) **108** (2017), 755–764, DOI: 10.1016/j.procs.2017.05.194.
- [67] Andy Nguyen, Ahmed E. Helal, Fabio Checconi, Jan Laukemann, Jesmin Jahan Tithi, Yongseok Soh, Teresa Ranadive, Fabrizio Petrini, and Jee W. Choi, *Efficient, Out-of-Memory Sparse MTTKRP on Massively Parallel Architectures* (en), Proceedings of the 36th ACM International Conference on Supercomputing, ICS '22, Association for Computing Machinery, Virtual Event, 2022, 1–13, DOI: 10.1145/3524059.3532363, arXiv:2201.12523 [cs].
- [68] D. Nion and N. D. Sidiropoulos, *Adaptive Algorithms to Track the PARAFAC Decomposition of a Third-Order Tensor*, IEEE Trans. Signal Process. **57** (2009), no. 6, 2299–2310.
- [69] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sri-ram Krishnamoorthy, and P. Sadayappan, *An Efficient Mixed-Mode Representation of Sparse Tensors*, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, Association for Computing Machinery, Denver, Colorado, 2019, DOI: 10.1145/3295500.3356216, event-place: Denver, Colorado.
- [70] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard Vuduc, and P. Sadayappan, *Load-Balanced Sparse MTTKRP on GPUs* (en), 2019 IEEE In-

- ternational Parallel and Distributed Processing Symposium (IPDPS), IEEE, Rio de Janeiro, Brazil, 2019, 123–133, DOI: 10.1109/IPDPS.2019.00023, DOI: 10.1109/IPDPS.2019.00023.
- [71] Alexander Novikov, Pavel Izmailov, Valentin Khrulkov, Michael Figurnov, and Ivan Oseledets, *Tensor train decomposition on TensorFlow (T3F)*, J. Mach. Learn. Res. **21** (2020), no. 1.
 - [72] Alexander Novikov, Dmitry Podoprikin, Anton Osokin, and Dmitry P. Vetrov, *Tensorizing Neural Networks*, Neural Information Processing Systems, 2015.
 - [73] Dianne P. O’Leary, *The block conjugate gradient algorithm and related methods*, Linear Algebra Appl. **29** (1980), 293–322, DOI: [https://doi.org/10.1016/0024-3795\(80\)90247-5](https://doi.org/10.1016/0024-3795(80)90247-5), Special Volume Dedicated to Alson S. Householder.
 - [74] I. Oseledets and Eugene E. Tyrtysnikov, *TT-cross approximation for multi-dimensional arrays*, Linear Algebra Appl. **432** (2010), 70–88.
 - [75] I. V. Oseledets, *Tensor-Train Decomposition* (en), SIAM J. Sci. Comput. **33** (2011), no. 5, 2295–2317, DOI: 10.1137/090752286.
 - [76] I. V. Oseledets and S. V. Dolgov, *Solution of Linear Systems and Matrix Inversion in the TT-Format*, SIAM J. Sci. Comput. **34** (2012), no. 5, A2718–A2739, DOI: 10.1137/110833142, eprint: <https://doi.org/10.1137/110833142>.
 - [77] Yannis Panagakis, Jean Kossaifi, Grigorios G Chrysos, James Oldfield, Michalis A Nicolaou, Anima Anandkumar, and Stefanos Zafeiriou, *Tensor Methods in Computer Vision and Deep Learning*, Proceedings of the IEEE **109** (2021), no. 5, 863–890, DOI: 10.1109/JPROC.2021.3074329.
 - [78] Evangelos E. Papalexakis, Christos Faloutsos, and Nicholas D. Sidiropoulos, *Tensors for Data Mining and Data Fusion: Models, Applications, and Scalable Algorithms*, ACM Trans. Intell. Syst. Technol. **8** (2016), no. 2, DOI: 10.1145/2915921, Place: New York, NY, USA Publisher: Association for Computing Machinery.
 - [79] Eric T. Phipps and Tamara G. Kolda, *Software for Sparse Tensor Decomposition on Emerging Computing Architectures*, SIAM J. Sci. Comput. **41** (2019), no. 3, C269–C290, DOI: 10.1137/18M1210691, Publisher: SIAM.

- [80] Achim Rettinger, Hendrik Wermser, Yi Huang, and Volker Tresp, *Context-Aware Tensor Decomposition for Relation Prediction in Social Networks*, Social Network Analysis and Mining **2** (2012), no. 4, 373–385, DOI: 10.1007/s13278-012-0069-5, ISBN: 1869-5469.
- [81] Amnon Shashua, Ron Zass, and Tamir Hazan, *Multi-way Clustering Using Super-Symmetric Non-negative Tensor Factorization*, European Conference on Computer Vision, Springer, 2006, 595–608.
- [82] N. Sidiropoulos and Ramakrishna S. Budampati, *Khatri-Rao space-time codes*, IEEE Trans. Signal Process. **50** (2002), 2396–2407.
- [83] Nicholas D. Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E. Papalexakis, and Christos Faloutsos, *Tensor Decomposition for Signal Processing and Machine Learning* (en), IEEE Trans. Signal Process. **65** (2017), no. 13, 3551–3582, DOI: 10.1109/TSP.2017.2690524, DOI: 10.1109/TSP.2017.2690524.
- [84] Shaden Smith, Alec Beri, and George Karypis, *Constrained Tensor Factorization with Accelerated AO-ADMM* (en), 2017 46th International Conference on Parallel Processing (ICPP), IEEE, Bristol, United Kingdom, 2017, 111–120, DOI: 10.1109/ICPP.2017.20.
- [85] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis, *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*, 2017.
- [86] Shaden Smith, Kejun Huang, Nicholas D. Sidiropoulos, and George Karypis, *Streaming Tensor Factorization for Infinite Data Sources*, Proceedings of the 2018 SIAM International Conference on Data Mining (SDM), SIAM, SIAM, 2018, 81–89, DOI: 10.1137/1.9781611975321.10.
- [87] Shaden Smith and George Karypis, *Tensor-Matrix Products with a Compressed Sparse Tensor*, Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, IA³ ’15, Association for Computing Machinery, Austin, Texas, 2015, DOI: 10.1145/2833179.2833183, event-place: Austin, Texas.
- [88] Shaden Smith and George Karypis, *Accelerating the Tucker Decomposition with Compressed Sparse Tensors* (en), Euro-Par 2017: Parallel Processing, Vol. 10417, Springer International Publishing, Cham, 2017, 653–668, DOI:

- 10.1007/978-3-319-64203-1_47, Series Title: Lecture Notes in Computer Science.
- [89] Shaden Smith and George Karypis, *SPLATT: The Surprisingly Parallel sparse Tensor Toolkit*, <https://github.com/ShadenSmith/splatt>, Online; accessed 14 May 2021, 2017.
 - [90] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis, *SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication*, 2015 IEEE International Parallel and Distributed Processing Symposium, IEEE, USA, 2015, 61–70, DOI: 10.1109/IPDPS.2015.27, DOI: 10.1109/IPDPS.2015.27.
 - [91] Yongseok Soh, Patrick Flick, Xing Liu, Shaden Smith, Fabio Checconi, Fabrizio Petrini, and Jee Choi, *High Performance Streaming Tensor Decomposition* (en), 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, Portland, OR, USA, 2021, 683–692, DOI: 10.1109/IPDPS49936.2021.00078.
 - [92] Yongseok Soh, Ahmed E. Helal, Fabio Checconi, Jan Laukemann, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jee W. Choi, *Dynamic Tensor Linearization and Time Slicing for Efficient Factorization of Infinite Data Streams*, 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, USA, 2023, 402–412, DOI: 10.1109/IPDPS54959.2023.00048, DOI: 10.1109/IPDPS54959.2023.00048.
 - [93] Yongseok Soh, Ramakrishnan Kannan, Piyush Sao, and Jee Choi, *Accelerated Constrained Sparse Tensor Factorization on Massively Parallel Architectures*, Proceedings of the 53rd International Conference on Parallel Processing, Association for Computing Machinery, New York, NY, USA, 2024, 107–116.
 - [94] Er Strehl, Joydeep Ghosh, and Raymond Mooney, *Impact of Similarity Measures on Web-page Clustering*, Workshop on Artificial Intelligence for Web Search (AAAI 2000) **58** (2001).
 - [95] Qingxiao Sun, Yi Liu, Ming Dun, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian, *SpTFS: Sparse Tensor Format Selection for MTTKRP via Deep Learning*, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20, IEEE Press, Atlanta, Georgia, 2020, 1–14, Place: Atlanta, Georgia.

- [96] Qingxiao Sun, Yi Liu, Hailong Yang, Ming Dun, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian, *Input-Aware Sparse Tensor Storage Format Selection for Optimizing MTTKRP*, IEEE Transactions on Computers **71** (2021), no. 8, 1968–1981, DOI: 10.1109/TC.2021.3113028.
- [97] Panagiotis Symeonidis, *Matrix and tensor decomposition in recommender systems*, Proceedings of the 10th ACM Conference on Recommender Systems, Association for Computing Machinery, New York, NY, USA, 2016, 429–430, DOI: 10.1145/2959100.2959195.
- [98] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J Leung, Manuel Egele, and Ayse K Coskun, *Diagnosing Performance Variations in HPC Applications using Machine Learning*, High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings 32, Springer, Springer, USA, 2017, 355–373, DOI: 10.1007/978-3-319-58667-0_19.
- [99] Melissa J. M. Turcotte, Alexander D. Kent, and Curtis Hash, *Unified Host and Network Data Set*, Data Science for Cyber-Security, World Scientific, 2018, chap. Chapter 1, 1–22, DOI: 10.1142/9781786345646_001.
- [100] Mark H. Van Benthem and Michael R. Keenan, *Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems* (en), Journal of Chemometrics **18** (2004), no. 10, 441–450, DOI: 10.1002/cem.889.
- [101] M. Vandecappelle, N. Vervliet, and L. De Lathauwer, *Nonlinear least squares updating of the canonical polyadic decomposition*, 2017 25th European Signal Processing Conference (EUSIPCO), 2017, 663–667.
- [102] Rama K Vasudevan, Maxim Ziatdinov, Vinit Sharma, Mark P Oxley, Lukas Vlcek, Anna N Morozovska, Eugene A Eliseev, Shi-Ze Yang, Yongji Gong, Pulickel Ajayan, et al., *Investigating phase transitions from local crystallographic analysis based on statistical learning of atomic environments in 2D MoS₂-ReS₂*, Applied Physics Reviews **8** (2021), no. 1.
- [103] N. Vervliet, O. Debals, and L. De Lathauwer, *Tensorlab 3.0 — Numerical optimization strategies for large-scale constrained and coupled matrix/tensor factorization*, 2016 50th Asilomar Conference on Signals, Systems and Computers, 2016, 1733–1738, DOI: 10.1109/ACSSC.2016.7869679, DOI: 10.1109/ACSSC.2016.7869679.

- [104] Samuel Williams, Andrew Waterman, and David Patterson, *Roofline: An Insightful Visual Performance Model for Multicore Architectures*, Commun. ACM **52** (2009), no. 4, 65–76, DOI: 10.1145/1498765.1498785, Place: New York, NY, USA Publisher: ACM.
- [105] Pranjul Yadav, Michael Steinbach, Vipin Kumar, and Gyorgy Simon, *Mining Electronic Health Records (EHRs) A Survey*, ACM Computing Surveys (CSUR) **50** (2018), no. 6, 1–40, DOI: 10.1145/3127881.
- [106] Yiming Yang and Bo Li, *nmf-torch: A Python package for Nonnegative Matrix Factorization with PyTorch*, Python Package Index, version 0.1.1, 2021.
- [107] Alaeddine Zahir, Khalide Jbilou, and Ahmed Ratnani, *High-dimensional multi-view clustering methods*, arXiv: 2303.08582, 2023.
- [108] Chundi Zheng, Meiyi Yu, Jiaolong Shan, Aiguo Wang, and Huihui Chen, *Fast Sparse Non-Negative Least Squares via ADMM for High Resolution DOA Estimation*, IEEE Sensors Journal **23** (2023), no. 4, 3901–3910, DOI: 10.1109/JSEN.2022.3233820.