SCALABLE TENSOR DECOMPOSITIONS IN HIGH PERFORMANCE COMPUTING ENVIRONMENTS

A Dissertation
Presented to
The Academic Faculty

By

Jiajia Li

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology

August 2018

SCALABLE TENSOR DECOMPOSITIONS IN HIGH PERFORMANCE COMPUTING ENVIRONMENTS

Approved by:

Dr. Richard W. Vuduc, Advisor
School of Computational Science & Engineering
Georgia Institute of Technology

Dr. Jimeng Sun
School of Computational Science & Engineering
Georgia Institute of Technology

Dr. Ümit V. Çatalyürek
School of Computational Science & Engineering
Georgia Institute of Technology

Dr. Tamara G. Kolda
Department of Data Science and Cyber Analytics
Sandia National Laboratories

Dr. Bora Uçar
LIP Computer Science laboratory
École normale supérieure de Lyon

Dr. David A. Bader
School of Computational Science & Engineering
Georgia Institute of Technology

Date Approved: July 25, 2018

There are no gains, without pains.

– *Benjamin Franklin*

To my parents and my fiancé.

ACKNOWLEDGEMENTS

First and foremost, I thank my advisor Rich Vuduc for being very helpful and supportive to my research, giving me freedom to pursue my own ideas, and offering constructive feedback. I really appreciate him for always backing me up, tolerating my countless shortcomings, and connecting me with good opportunities. He was also very considerate to my life and health. He has been a great role model. I have learned a lot from him, his kindness, his open mindedness, his insistence on academic rigor, and his passion for good food.

I also thank other faculty at Georgia Tech. Jimeng Sun has been deeply involved in my research, giving his valuable time and advice, particularly on applications of tensors. He made me feel my work has the potential for meaning and impact. I also thank him for his terrific advice on my career path. I am grateful to David Bader for working together with and advising me during my first year which is my most "ridiculous" time. I appreciate his encouragement and career suggestions which helped me to become a better person. I thank Ümit Çatalyürek, Edmond Chow, Haesun Park, and Srinivas Aluru for taking their time to serve on my academic committees. I also thank Will Powell, Arlene Washington-Capers, Nirvana Edwards, Carolyn Young, Anna Stroup, Della Phinisee, and Mimi Haley for their kind and timely help on annoying "chores."

External to Georgia Tech, I am very grateful to especially Tamara Kolda, Bora Uçar, Guangming Tan, Xipeng Shen, Chunhua Liao, Chenggang Yan, Dong Chen, Mikhail Smelyanskiy, and David Keyes. Tamara Kolda relayed her many experiences and stories, which encouraged me a lot, along with her cute GT scarf gift. I appreciate that she always introduced me to new people and exposed me to different ideas and areas of work. She is also a great role model to me, being always organized, friendly, and showing strong leadership. Bora Uçar is an amazing researcher who contributes tremendously in our collaborated work. I really appreciate his effort and had a wonderful time to working with him. Guangming Tan is a good friend to me, who still put faith on me after these many years and dragged me

being involved into diverse research topics. I am grateful to Xipeng Shen and Chuanhua Liao for giving me a chance to continue working on some of my previous topics. I really enjoyed working with them. I also thank Chuanhua for his valuable advice on my career path and life plan. I thank Chenggang Yan for collaborating with me and putting his undergraduate students under my guidance, which gave me good experience in advising. I am very grateful to Dong Chen, my group manager in IBM Thomas J. Watson Research Center, who was very supportive for my career. I also thank Mikhail Smelyanskiy, my supervisor in Intel Parallel Computing Lab, who helped increase my knowledge of architecture-aware optimization techniques. A special thank to David Keyes, your applause in a SC workshop for my five-minutes presentation encouraged me a lot and reminded me of being able to do good presentations.

I am grateful to my colleague collaborators: Jee Choi, Xing Liu, Shaden Smith, Casey Battaglino, Ioakeim (Kimis) Perros, Srinivas Eswar, Patrick Lavin, Wafa Louhichi, Peter James Ahrens, Yue Zhao, Junhong Liu, Xiuxia Zhang, Keren Zhou, Zhonghai Zhang for their help to make the research great. I thank my lab mates Marat Dukhan, Oded Grean, Piyush Sao, Patrick Flick, Michael Isaev, Robert Chen, Sara Karamati, Bo Dai, Eisha Nathan, SaBra Neal, Lluis Munguia, Elias Khalil, Xiaojing An, Chunxing Yin, Kasimir Gabert, Cong Chen, and Jing Shui, without you guys the Ph.D. life will not be such enjoyable.

I was fortunate to have worked with a couple of great undergraduate researchers, especially Yuchen Ma, Nick Liu, and Junghyun Kim. I learned both knowledge and advising experience from this process. I specially thank Yuchen Ma for helping build the infrastructure of PARTI! library.

Finally, I owe my deepest gratitude to my family. My mom supported me as always, encouraged me, pointed me out the most appropriate direction, and cheered me up with her never-ending love. I give the full credit of my good health after the second Ph.D. program to my mom, who traveled to the U.S. on several occasions to take care of and feed me, and

her continually nudges to stay on-track EVERYDAY.

I am also grateful to my fiancé, Xiaolong, who has accompanied me to complete my two Ph.D. programs. I am sorry I made him suffer from time to time over these last few years because of my "endless" bad temper, of which he often ended up being the target. Despite that, he remained always supportive, understanding (sometimes even more than my mom), and being the best listener. I could not have gotten this far without him.

This research is supported by the U.S. National Science Foundation (NSF) Award Number 1533768, 2017–2018 IBM Ph.D. Fellowship Award, and the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of NSF, IBM, or Sandia National Laboratories.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This dissertation presents novel algorithmic techniques and data structures to help build scalable tensor decompositions on a variety of high-performance computing (HPC) platforms, including multicore CPUs, graphics co-processors (GPUs), and Intel Xeon Phi processors.

A tensor may be regarded as a multiway array, generalizing matrices to more than two dimensions. This dissertation conquers some challenges of tensor algorithms: *the curse of dimensionality, mode orientation, tensor transformation, irregularity, and arbitrary tensor dimensions (or orders)*, in the specific context of the two of the most popular tensor decompositions, the CANDECOMP/PARAFAC (CP) and Tucker decompositions, which are, roughly speaking, the tensor analogues to low-rank approximations in standard linear algebra. Within that context, two of the critical computational bottlenecks are the operations known as Tensor-Times-Matrix (TTM) and Matricized Tensor Times Khatri-Rao Product (MTTKRP). We consider these operations in cases when the tensor is dense or sparse.

ADATM work is the application of memoization to overcome the curse of dimensionality challenge that exists in a sequence of tensor operations. By saving and reusing the intermediate results of MTTKRPs, our method also reduces the overall floating-point operations and memory transfers, with increasing benefits as the order of the tensor increases. Our proposed algorithm is parameterized in a way that may be tuned for performance given a specific hardware platform.

A novel data structure, hierarchical coordinate format (HICOO), is proposed to address the challenge of mode orientation for sparse tensors. Mode orientation refers to the concept of a particular format favoring iteration of tensor modes. By compressing the indices in units of sparse tensor blocks, with the goals of preserving the "mode-agnostic" simplicity of coordinate (COO) format while reducing the bytes needed to represent the tensor and promoting data locality. A parallel scheduler to avoid the locks for write-conflict memory

is also proposed. CP decomposition using HICOO format achieves considerable speedups over COO and another state-of-the-art compressed sparse fiber (CSF) format. We also propose reordering methods to arrange non-zeros for denser tensor blocks to pursue higher performance.

By avoiding tensor-matrix conversions, we are able to carry out TTM operation in-place, for dense and sparse cases. INTENSLI uses loops of matrix-matrix multiplications (GEMM) to implement dense TTM in-place and choose an appropriate matrix shape to achieve high performance. The computation of the sparse TTM operation is implemented on all non-zero elements with no explicit conversion. Sparsity is challenging because it induces irregular memory access. We employ optimization techniques such as increasing memory coalescing, dynamic parallelism, and ranking blocking, among others, in our GPU implementations.

To validate these ideas, we have implemented them in three prototype libraries, named ADATM, INTENSLI, and PARTI!, for arbitrary-order tensors. ADATM is a model-driven framework to generate an adaptive tensor memoization algorithm with the optimal parameters for sparse CP decomposition. INTENSLI produces fast single-node implementations of dense TTM of an arbitrary dimension. PARTI! is short for a Parallel Tensor Infrastructure which is written in C, OpenMP, MPI, and NVIDIA CUDA for sparse tensors and supports MATLAB interfaces for application-level users.

CHAPTER 1
INTRODUCTION

Tensors, which generalize matrices to more than two dimensions, have found numerous uses in data analysis and mining. In particular, tensor-based analysis methods have been noted for their ability to discover multi-dimensional dependencies [3, 4, 6, 44, 46, 50, 51, 54, 89, 111, 123, 131, 134, 150], and has been applied to healthcare analytics [75, 175], social networks analytics [129, 130], computer vision [169], natural language processing [80], signal processing [6], and neuroscience [98], to name a few. However, while the number of potential uses of tensor methods has grown, much of the available software implementing those methods is not yet capable of fully exploiting the computing potential of high-performance computing (HPC) platforms, such as multicore CPUs, graphics co-processors (GPUs), and Intel Xeon Phi processors. Such a capability is needed to apply tensor methods at scale.

When attempting to implement tensor algorithms efficiently on HPC platforms, there are several obstacles. This dissertation is particularly concerned with five such issues, which we will refer to as *the curse of dimensionality*, *mode orientation*, *tensor transformation*, *irregularity*, and *arbitrary tensor dimensions* (or orders). These challenges bring non-trivial computational and storage overheads. Some of them, such as the curse of dimensionality, tensor transformation, and arbitrary tensor dimensions (or orders), are particular to tensor methods; while other challenges, such as mode orientation and irregularity, are even harder to overcome than their counterparts in classical linear algebra, i.e., matrix decompositions.

The overarching goal of this dissertation is to develop techniques that overcome these challenges. Our approach is to use algorithmic and data structure changes combined with better engineering of the implementations. We validate our ideas by realizing these tech-

niques in several prototypes of high-performance libraries, rigorously evaluated on modern parallel platforms. For concreteness, we work in the context of what are arguably the two of the most popular tensor decompositions, namely, CANDECOMP/PARAFAC (CP) and Tucker decompositions, both of which can produce low-rank representations of data. Their two main computational bottlenecks are known as Tensor-Times-Matrix (TTM) and Matricized Tensor Times Khatri-Rao Product (MTTKRP). We consider large-scale dense and sparse tensors with a focus on sparse tensors because of their wide use in numerous real-world data.

In the remainder of this section, we briefly explain the challenges and provide pointers into the dissertation where we consider them.

The curse of dimensionality refers to the issue that the number of entries of an intermediate or output tensor can grow exponentially with the tensor order, resulting in significant computational and storage overheads. Even when the tensor is structurally sparse, meaning it consists mostly of zero entries, the execution time of one important tensor method, the so-called CP decomposition considered elsewhere in this dissertation, generally grows quadratically with the number of non-zeros [15, 16]. And there is an increasing interest in applications involving a large number of dimensions [55, 99, 123]. While there is some current research looking at this challenge in the context of Khatri-Rao products in CP for dense tensors [137, 190], we will be particularly interested in the sparse case (Chapter 3).

Mode orientation refers to the issue of a particular storage format favoring the iteration of tensor modes in a certain sequence, which is of particular concern in the sparse case. Since most methods of interest require more than one sequence, being efficient for every sequence generally requires storing the tensor in multiple formats, thereby trading extra memory for speed. Today, three commonly used formats are coordinate (COO) [15], compressed sparse fibers (CSF) [152], and flagged coordinate (F-COO) [107] formats. While COO is considered to have a neutral mode orientation, meaning it does not favor any particular iteration sequence, CSF and F-COO do; however, COO is also larger in size than

CSF and F-COO to begin with. A natural question arises, which is whether one can achieve both a neutral mode orientation and compact storage. We propose one such scheme, which we call *hierarchical coordinate* (HICOO) format (Chapter 4).

Tensor transformation(s) refers to a common pattern for attaining speed in some implementations of tensor algorithms, which starts by reorganizing the tensor into a matrix and then perform equivalent matrix operations using highly tuned linear algebra libraries. Done naïvely, this approach appears to require an extra memory copy, which can even come to dominate the overall running time. We observe instances in which such a copy consumes 70% or more of the total running time (in the case of a TTM operation). We devise ways to avoid such transformations using in-place alternative formulations, specifically for TTM and MTTKRP operations for dense (Chapter 6) and sparse cases (Chapter 7).

Irregularity refers to two issues. The first is that a tensor may have dimension sizes that vary widely; the second is that a sparse tensor may have an irregular non-zero pattern, resulting in irregular memory references. We consider the problem of irregular tensor shape in a dense TTM operation (Chapter 6), sparse MTTKRP (Chapter 4), and sparse TTM operation on GPUs (Chapter 7), using a variety of algorithmic reorganization, parallelization, and performance engineering strategies.

Arbitrary tensor orders generate various implementations of a tensor operation. For the sake of performance, programmers usually implement and optimize third-order tensor algorithms apart from higher-order ones. These implementations makes no one optimization method can fit all variations, e.g., different number of loops and diverse memory access behavior. One optimization usually behaves well for a fixed, traceable implementation such as matrix-vector multiplication and other matrix operations. All of our work in this dissertation supports arbitrary order of tensors and flexibly chooses an optimal implementation for a particular tensor order if possible.

1.1 Contributions

We briefly summarize the main contributions of this dissertation as follows.

- We propose a novel, adaptive tensor memoization algorithm, which we refer to as AdaTM[1]. Our proposed method reduces the flop-complexity of CPD from prior implementations [15, 41, 80, 84, 156], which scales better as the tensor order grows. We develop a model-driven framework to tune several parameters of our method and determine their time and space tradeoff. (See Chapter 3, which is published [101].)

- We propose a novel sparse tensor format, *hierarchical coordinate*, or HiCOO[2], that tries, heuristically, to achieve both compactness and neutral mode orientation. HiCOO compresses tensor indices in units of sparse tensor blocks and exploits shorter integer types to express offsets within the blocks. Z-Morton order is employed between and inside tensor blocks to further improve data locality. A superblock scheduler and two parallelization strategies are used to accelerate MTTKRP and CPD on multicore CPUs based on HiCOO. We also employ reordering methods to obtain better data locality for HiCOO. (See Chapter 4, which is accepted for publication [104].)

- We present a novel framework InTensLi[3] for automatically generating high-performance in-place dense TTM implementations. It considers several strategies for decomposing the specific type of TTM that the user requires in order to maximize the use of the fast GEMM. The code generation framework produces several parameterized implementations; and we use a heuristic model to select the parameters, which need to be tuned for a given input. (See Chapter 6, which is published [100].)

- We optimize tensor-times-dense matrix multiply (TTM) and Tucker decomposition

[1] Read, "Ada-Tee-Em," as in Ada Lovelace.
[2] pronounced "haiku"
[3] pronounced as "intensely"

for general sparse and semi-sparse tensors on multicore CPUs and NVIDIA GPUs. An in-place sequential SpTTM algorithm is designed to avoid tensor-matrix conversion. We propose several optimizing approaches for SpTTM on NVIDIA GPUs and implement SspTTM on GPUs accordingly by better exploring the dense substructures. A faster Tucker decomposition is obtained by integrating the optimized SpTTM and SspTTM. (See Chapter 7, which is accepted for publication [113].)

Several of these ideas are now combined in a software library named ParTI!, which is short for "parallel tensor infrastructure." (See Chapter 8, which is partially published [102]) It intends to support high-performance CP and Tucker decompositions on several platforms, including multicore CPUs, GPUs, distributed systems, and even a new architecture, the Emu. At present, it includes support for two sparse tensor formats, the popular COO and our new HiCOO. With ParTI!, we aim to hasten the translation of the ideas of this dissertation into analysis practice.

CHAPTER 2

BACKGROUND ON TENSORS AND PARALLEL PLATFORMS

A tensor, abstractly defined, is a function of three or more indices. In computational data analytics, one may regard a tensor as a multidimensional array, where each of its dimensions is also called a *mode* and the number of dimensions or modes is its *order*. For example, a scalar is a tensor of order 0; a vector is a tensor of order 1; and a matrix, order 2, with two modes (its rows and its columns). Notationally, we represent tensors as calligraphic capital letters, e.g., $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$; matrices by boldface capital letters, e.g., $\mathbf{U} \in \mathbb{R}^{I \times J}$; vectors by boldface lowercase letters, e.g., $\mathbf{x} \in \mathbb{R}^{I}$; and scalars by lowercase letters, such as x_{ijk} for the (i, j, k) element of a third-order tensor $\mathcal{X}$. A third-order tensor is illustrated in figure 2.1(a).

A tensor may be *dense*, meaning all or most of its entries are non-zero; or *sparse*, meaning most of its entries are zeros. Sparsity implies that one may implement operations on the tensor using a compact data structure that avoids storage of and operations on these zeros. Figure 2.1(b) uses solid dots to show non-zero entries of a sparse third-order tensor. For a more complete description see Kolda and Bader's survey [89]. Table 2.1 lists the general symbols and notations used in this dissertation. More specific symbols will be shown separately in each chapter below.

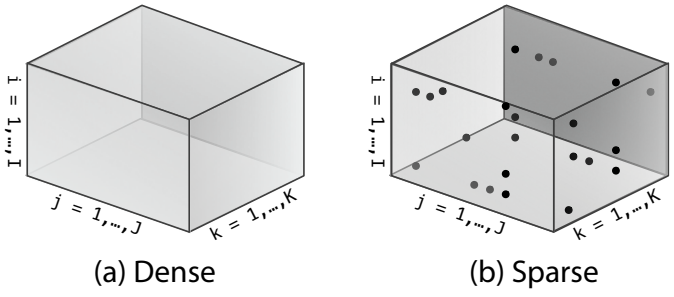


Figure 2.1: Visualization of dense and sparse third-order tensors $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$.

Table 2.1: List of general symbols and notations in this dissertation.

Symbols	Description
Word size parameters	
β_{byte}	Bit-length of a byte or character
β_{int}	Bit-length of an integer
β_{long}	Bit-length of a long integer
β_{float}	Bit-length of a single-precision floating point value
β_{double}	Bit-length of a double-precision floating point value
Data representation	
$\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{G}$	Tensors
$\mathbf{X}_{(n)}$	Matricized tensor $\mathcal{X}$ in mode-n
$\mathbf{A}, \tilde{\mathbf{A}}, \mathbf{U}$	Dense matrices
$\mathbf{a}, \mathbf{a_r}$	Dense vectors
$x_{i_1 \ldots i_N}$	A tensor entry
$\boldsymbol{\lambda}$	Weight vector
Input parameters	
N	Tensor order
I, J, K, L, I_n	Tensor mode sizes
M_O	Tensor mode order
N_D, N_S	#Dense modes and #Sparse modes
M	#Nonzeros of the input tensor $\mathcal{X}$
M_l	#Nodes at level l of a CSF tree of tensor $\mathcal{X}$
R, R_n	Approximate tensor ranks (usually a small value)
Operations	
$\mathbf{a} \circ \mathbf{b}$	Outer product of two vectors
$\mathbf{a} * \mathbf{b}$	Hadamard or element-wise product of two vectors
$\mathbf{A} \otimes \mathbf{B}$	Kronecker product of two matrices
$\mathbf{A} \odot \mathbf{B}$	Khatri-Rao product of two matrices
$\mathcal{X} \oplus \mathcal{Y}$	Element-wise addition of two tensors
$\mathcal{X} \times_n \mathbf{U}$	Mode-n tensor-times-matrix product (TTM)
$\mathbf{X}_{(n)}(\odot_{i=1,\ldots,N}^{i \neq n} \mathbf{U}_i)$	mode-n Matriced Tensor Times Khatri-Rao Product (MTTKRP)
COO Data structure	
inds	A two-level index array of COO, β_{int} bits
val	A non-zero value array of COO, β_{float} or β_{double} bits

2.1 Tensor Representations

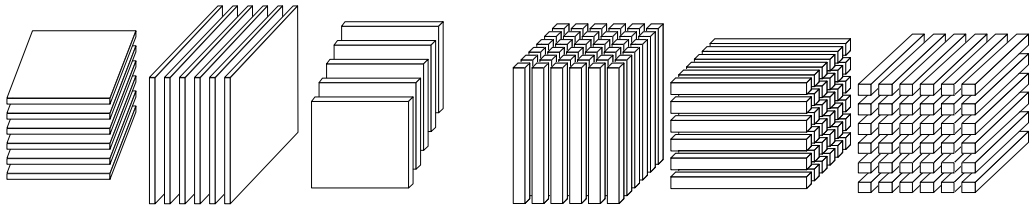
There are different ways to view a tensor logically, as well as different ways of storing tensor data in a concrete program. We will distinguish between the logical or *mathematical* representations (Section 2.1.1) and "physical" data structures (Sections 2.1.2 to 2.1.3), as described next.

2.1.1 Mathematical Representations

Frequently, we will take a high-order tensor and extract or operate on some lower-order object. For instance, we might wish to *reshape* or *fold* all the elements of a tensor of order 3 or higher into a matrix or a vector; these operations are called *matricization* or *vectorization*, respectively. Alternatively, we might ask for a subset of the elements of a tensor in the form of a matrix or a vector; these are called *slices* or *fibers*, respectively. All of these representations are "logical" or "mathematical" in the following sense: we might describe some tensor operation in terms matricized or vectorized tensors or slices or fibers, but the implementation of that operation need not involve a physical reorganizing of elements in memory.

Given a tensor, there are multiple ways to select a *sub-tensor*. A *slice* is a two-dimensional cross-section of a tensor, achieved by fixing all mode indices but two, e.g., $\mathbf{S}_{::k} = \mathcal{X}(:,:,k)$ in MATLAB notation. A *fiber* is a vector extracted from a tensor along some mode, selected by fixing all indices but one, e.g., $\mathbf{f}_{:jk} = \mathcal{X}(:,j,k)$. They are illustrated in figure 2.2 for a third-order tensor. We will make heavy use of sub-tensors when optimizing the TTM operation in Chapter 6.

Matricization, or unfolding, is the process of flattening a tensor into an equivalent matrix. Some subset of the modes constitute the matrix row, while the remaining modes constitute the matrix column. In the mode-n matricization of $\mathcal{X}$, which we denote by $\mathbf{X}_{(n)} \in \mathbb{R}^{I_n \times I_1 \cdots I_{(n-1)} I_{(n+1)} \cdots I_N}$, the fibers along the single mode n are stacked along the



(a) Horizontal, lateral, and frontal slices

(b) Column (mode-1), row (mode-2), and tube (mode-3) fibers

Figure 2.2: Slices and fibers of a third-order tensor.

columns. The order in which they are stacked may use the lexicographic ordering of the remaining modes, or any other consistent ordering. We may also want to matricize along multiple modes, e.g., $\mathcal{X}_{[I_1 I_2]\times[I_3 I_4 \cdots I_N]}$ stacks vectors formed from the first two modes along the columns. *Vectorization*, $\mathbf{y} = \text{vec}(\mathcal{X})$, refers to the conversion of a tensor into a vector. Vectorization can be implemented by concatenating tensor fibers, again in some consistent order. The reverse process of creating a tensor from a matrix or vector is called *tensorization*, and it appears in some applications to extract more hidden information [55, 99, 123]. A third-order example of mode-2 matricization and its reverse tensorization process are shown in figure 2.3.

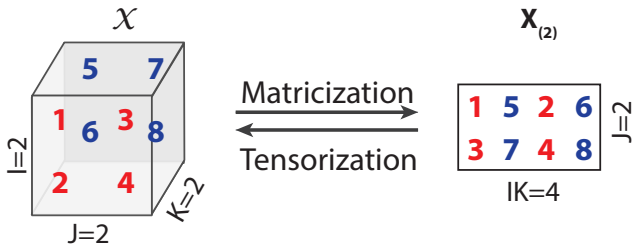


Figure 2.3: Mode-2 matricization of an example third-order tensor and its tensorization.

2.1.2 Dense Tensor Layout

Analogous to the way dense matrices may be stored in row-major or column-major conventions under different language environments or user specifications, we need some conventions for laying out a dense tensor in memory. We will use a *mode order descriptor*, $M_O = \text{permute}[1, \ldots, N]$, to indicate the layout. For instance, consider a third-order ten-

sor stored in the mode order, $M_O = [3, 2, 1]$. The tensor elements that are contiguous in mode 3 will be immediately adjacent in memory, followed by mode 2, and lastly mode 1. This example appears as "Last-mode dominated" in figure 2.4. A third-order tensor has 6 distinct data layouts, $[3, 2, 1]$, $[3, 1, 2]$, $[2, 3, 1]$, $[2, 1, 3]$, $[1, 3, 2]$, $[1, 2, 3]$, where $[3, 2, 1]$ and $[1, 2, 3]$ are the most common *last-* and *first-mode dominated* layouts, respectively. Figure 2.4 shows these two layout examples for a third-order tensor.

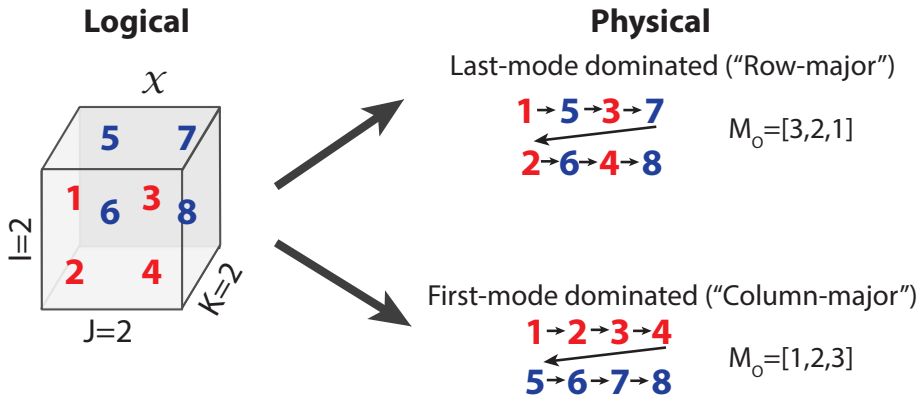


Figure 2.4: Dense tensor layout examples.

Different data layouts affect the performance of a tensor algorithm, depending on what order it uses to iterate over the tensor's elements. If an algorithm's memory access pattern matches the tensor's layout, meaning that the data access in the algorithm has good spatial locality, we may observe relatively higher performance than if not; otherwise, the data access is in a strided pattern (at a possibly large stride) may be relatively slower. We study the influence of data layout for the tensor-times-matrix multiplication (TTM) operation in Chapter 6.

2.1.3 Sparse Tensor Formats

In data-oriented applications of tensors, the tensor object is often extremely sparse. Therefore, most of this dissertation concerns the sparse case.

Analogous to the dense case, there is a considerable body of work on efficient implementation of operations on *sparse matrices* [13, 30–33, 43, 93, 105, 110, 120, 172, 173,

178, 180, 181, 187]. As with sparse matrices, we will need to consider how to store a sparse tensor to minimize storage and computational overheads.

This dissertation will be especially concerned with general sparse tensors, for which we will make few assumptions about numerical symmetry or other pattern structure. For a tensor symmetric along some mode, we might expect storage to go down by half [20, 158, 159]; and a sparse tensor with dense blocks could be compressed to a smaller much one with each element holding a dense block [34, 50, 51, 54, 60, 127, 148]. Instead, this dissertation will take as its baseline representations three state-of-the-art sparse tensor formats, COO, CSF, and F-COO, summarized below.

Coordinate, or COO, format. COO is the simplest and arguably de facto standard way to store a sparse tensor. We use inds and val to represent the indices and values of the non-zeros of a sparse tensor respectively (listed in table 2.1). val is a size-M array of floating-point numbers, inds is a size-M array of integer tuples. Figure 2.5(a) shows a $4 \times 4 \times 3$ sparse tensor in COO format. The indices of each mode are represented as i, j, and k. Observe that some indices in inds repeat, for example, entries $(1, 0, 0)$ and $(1, 0, 2)$ have the same i and j indices. This redundancy suggests some compression of this indexing metadata should be possible.

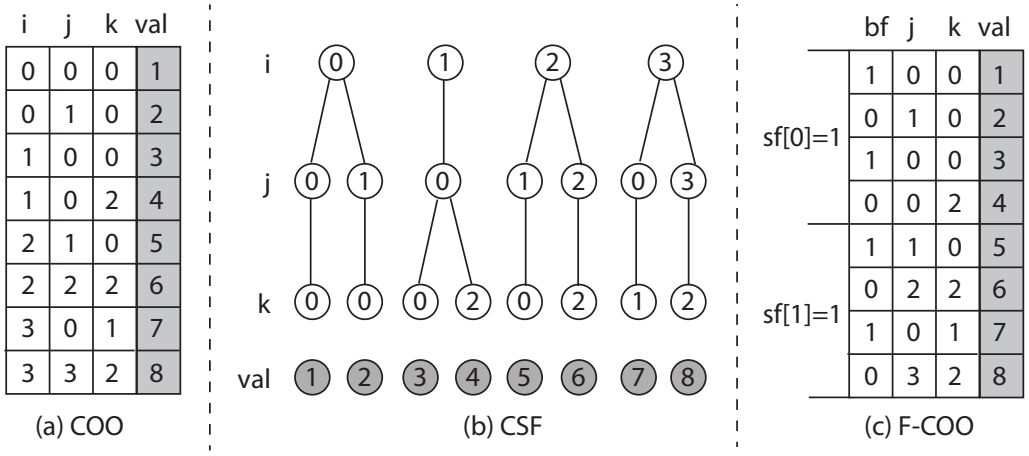


Figure 2.5: Sparse tensor formats of an example $4 \times 4 \times 3$ tensor. This CSF tree and F-COO representation are both for mode order $M_O = [1, 2, 3]$.

Compressed sparse fiber, or CSF, format. The Compressed Sparse Fiber (CSF) format is a hierarchical, fiber-centric format that generalizes the popular Compressed Sparse Row (CSR) format from sparse matrices to tensors [152, 154, 156]. It is memory-efficient and exhibits good speedups on matricized tensor times Khatri-Rao product (MTTKRP) operation (see § 2.2) compared to a COO-based implementation.

CSF format is *mode-oriented*, by which we mean that distinct mode orders (M_Os) lead to different CSF representations. A CSF representation with a given M_O is called a ***CSF tree***. Figure 2.5(b) shows a CSF tree with $M_O = [1, 2, 3]$. This CSF tree takes indices in mode 1 as the root nodes, indices in mode 2 as the internal nodes, and indices in mode 3 as the leaf nodes. Observe that there is compression wherever there are shared nodes (indices). For instance, entries $(1, 0, 0)$ and $(1, 0, 2)$ share the same internal and root nodes.

Since CSF format places the modes in some sequence, tensor operations based on CSF format exhibit mode-dependent performance behavior. In particular, if a tensor algorithm requires iterating over the modes in a sequence that differs from the sequence implied by a root-to-leaf path in the CSF tree, then one may expect inefficient memory access and duplicated computation to occur, thereby reducing algorithmic performance compared to an iteration sequence that matches that of the CSF tree. The naïve way to avoid such inefficiencies are to maintain copies of the CSF tree in multiple mode orders, at the price of extra storage.

Flagged-coordinate, or F-COO, format. A recently proposed Flagged-COOrdinate (F-COO) format uses a different compression idea [107], which is inspired by segmented scan operations [25]. In particular, it maintains two bit-arrays, "bit-flag" (bf) and "start-flag" (sf), to replace the index mode(s), while product mode(s) and their indices are left unchanged. The bf represents changes in the index mode(s) which consequently separates independent computations. The sf indicates whether the segments processed by the current thread start new indices of the index mode(s). An example F-COO representation for MT-

TKRP is shown in figure 2.5(c) with $M_O = [1, 2, 3]$. F-COO format relies not only the mode order but also the particular tensor operation. Therefore, multiple F-COO representations may be needed for different tensor operations and for the same operation when operating on a different mode.

CSF and F-COO formats speed up tensor operations over COO format, but they show mode-oriented behavior. We propose a hierarchical coordinate (HICOO) format with neutral mode-orientation to improve this issue in Chapter 4.

2.2 Fundamental Tensor Operations

There are a variety of computational operations we will perform on and using tensors. The most important are *products*, meaning the tensor analogues of matrix products (multiplication) in conventional linear algebra.

In describing these operations, there is some additional terminology we will use throughout to distinguish specific properties of a tensor's modes:

- *Product mode(s)*: the mode(s) in which a tensor is "joined" with another tensor during a product operation. For example, when multiplying a matrix $\mathbf{A}$ by another $\mathbf{B}$, the product mode of $\mathbf{A}$ is its columns, which must be joined with the product mode of $\mathbf{B}$, its rows.
- *Index mode(s)*: the remaining mode(s) excluding the product mode(s). In the $\mathbf{A} \cdot \mathbf{B}$ example, the rows of $\mathbf{A}$ and the columns of $\mathbf{B}$ are the index modes.
- *Dense mode(s)*: the mode(s) in which all non-empty fibers are fully dense.
- *Sparse mode(s)*: the mode(s) in which at least one non-empty fiber is sparse.

2.2.1 Element-wise Operations

Element-wise operations include addition, subtraction, multiplication, and division operations, which are applied to every corresponding pair of elements from two tensor ob-

jects having the same order and mode sizes. For example, element-wise tensor addition of $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is $\mathcal{Z} = \mathcal{X} \oplus \mathcal{Y}$, where

$$z_{i_1 \cdots i_N} = x_{i_1 \cdots i_N} + y_{i_1 \cdots i_N}. \tag{2.1}$$

Element-wise operations can be easily implemented by iterating all non-zeros of the two sparse tensors.

2.2.2 Tensor-Times-Matrix

The Tensor-Times-Matrix (TTM) in mode n, also known as the n-mode product, is the multiplication of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ with a matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, along mode n, and is denoted by $\mathcal{Y} = \mathcal{X} \times_n \mathbf{U}$.[1] This results in a $I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \times \cdots \times I_N$ tensor, and its operation is defined as

$$y_{i_1 \cdots i_{n-1} r i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 \cdots i_{n-1} i_n i_{n+1} \cdots i_N} u_{i_n r}. \tag{2.2}$$

TTM is a special case of tensor contraction. We consider TTM specifically because of its more common usage in tensor decompositions for data analysis, such as the Tucker decomposition. Also, note that R is typically much smaller than I_n in such decompositions, and typically $R \approx 100$.

TTM is also equivalent to a matrix-matrix multiplication in the following form:

$$\mathcal{Y} = \mathcal{X} \times_n \mathbf{U} \quad \Leftrightarrow \quad \mathbf{Y}_{(n)} = \mathbf{U}\mathbf{X}_{(n)}. \tag{2.3}$$

Therefore, a practical way to implement an efficient TTM is to first matricize the tensor, then use an optimized matrix-matrix multiplication to compute the matricized output $\mathcal{Y}$,

[1] Our convention for the dimensions of $\mathbf{U}$ differs from that of Kolda and Bader's definition [89]. In particular, we transpose the matrix modes $\mathbf{U}$, which leads to a more efficient TTM under the row-major storage convention of the C language.

and, finally, tensorize to obtain $\mathcal{Y}$.

2.2.3 Kronecker and Khatri-Rao Products

The *Kronecker product* generalizes the outer product for matrices. Given $\mathbf{U} \in \mathbb{R}^{I \times J}$ and $\mathbf{V} \in \mathbb{R}^{K \times L}$, the Kronecker product $\mathbf{U} \otimes \mathbf{V} \in \mathbb{R}^{IK \times JL}$ is

$$\mathbf{U} \otimes \mathbf{V} = \begin{bmatrix} u_{11}\mathbf{V} & u_{12}\mathbf{V} & \cdots & u_{1J}\mathbf{V} \\ u_{21}\mathbf{V} & u_{22}\mathbf{V} & \cdots & u_{2J}\mathbf{V} \\ \vdots & \vdots & \ddots & \vdots \\ u_{I1}\mathbf{V} & u_{I2}\mathbf{V} & \cdots & u_{IJ}\mathbf{V} \end{bmatrix}$$

The *Khatri-Rao product* is a "matching column-wise" Kronecker product between two matrices with the same number of columns. Given matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$, their Khatri-Rao product is denoted by $\mathbf{A} \odot \mathbf{B} \in \mathbb{R}^{(IJ) \times R}$,

$$\mathbf{A} \odot \mathbf{B} = \left[\mathbf{a}_1 \otimes \mathbf{b}_1, \mathbf{a}_2 \otimes \mathbf{b}_2, \ldots, \mathbf{a}_R \otimes \mathbf{b}_R\right], \tag{2.4}$$

where $\mathbf{a}_r$ and $\mathbf{b}_r$, $r = 1, \ldots, R$, are columns of $\mathbf{A}$ and $\mathbf{B}$.

Kronecker and Khatri-Rao products appear frequently in tensor decompositions that are formulated as matrix operations. However, such formulations typically also require redundant computation or extra storage to hold matrix operands, so in practice these operations are not implemented directly but rather integrated into other operations.

2.2.4 Matriced Tensor Times Khatri-Rao Product

For an N^{th}-order tensor $\mathcal{X}$ and given matrices $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$, the mode-$n$ MTTKRP is

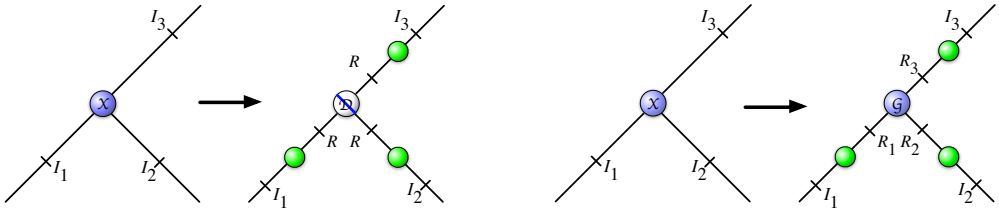
$$\tilde{\mathbf{U}}^{(n)} = \mathbf{X}_{(n)} \left(\odot_{i=1,\ldots,N}^{i \neq n} \mathbf{U}_i\right) = \mathbf{X}_{(n)} \left(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(n+1)} \odot \mathbf{U}^{(n-1)} \odot \cdots \odot \mathbf{U}^{(1)}\right), \tag{2.5}$$

where $\mathbf{X}_{(n)}$ is the mode-n matricization of tensor $\mathcal{X}$, $\odot$ is the Khatri-Rao product. The output $\tilde{\mathbf{U}}^{(n)}$ is used to update the matrix operand $\mathbf{U}^{(n)}$ for the next MTTKRP in a ten-

sor decomposition. MTTKRP is a critical computational kernel of a popular CANDECOMP/PARAFAC decomposition.

2.3 Tensor Decompositions and Algorithms

This dissertation is specifically interested in efficient computation of low-rank tensor decompositions, the two most popular ones being the CANDECOMP/PARAFAC (CP) and Tucker decompositions. These models produce, as their main outputs, dense matrices.



(a) CP decomposition (b) Tucker decomposition

Figure 2.6: CP and Tucker decompositions illustrated using a tensor networks diagram [44]. Each node in a tensor network corresponds to a tensor with each incident edge represents a mode of it. Each edge between nodes corresponds to a product mode that will be contracted upon, and each unconnected edge corresponds to an index mode.

2.3.1 CP Decomposition

The CANDECOMP/PARAFAC decomposition (CPD) [37, 73] decomposes a tensor into a sum of component rank-one tensors, shown in figure 2.6(a). It approximates an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ as

$$\mathcal{X} \approx \sum_{r=1}^{R} \lambda_r \mathbf{u}_r^{(1)} \circ \cdots \circ \mathbf{u}_r^{(N)} \equiv [\![\boldsymbol{\lambda}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!], \tag{2.6}$$

where R is the canonical rank of tensor $\mathcal{X}$, taken as the number of component rank-one tensors [89]. In a low-rank approximation, R is usually chosen to be a small number less than 100. The outer product of the vectors $\mathbf{u}_r^{(1)}, \ldots, \mathbf{u}_r^{(N)}$ produces R rank-one tensors,

and $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R}, n = 1, \ldots, N$ are the *factor matrices*, each one formed by taking the corresponding vectors as its columns, i.e., $\mathbf{U}^{(n)} = [\mathbf{u}_1^{(n)}\ \mathbf{u}_2^{(n)} \ldots \mathbf{u}_R^{(n)}]$. We normalize these vectors to unit magnitude and store the factor weights in the vector $\boldsymbol{\lambda} = \{\lambda_1, \ldots, \lambda_r\}$. The format $[\![\boldsymbol{\lambda}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$ is called a Kruskal tensor [90], which is used in the discovery and interpretation of latent relationships in data analytics. For an Nth-order hypercubical tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times \cdots \times I}$, its factored Kruskal tensor only needs $\mathcal{O}(NRI)$ parameters (or entries) which is smaller than $\mathcal{O}\big(I^N\big)$ if it is dense or $\mathcal{O}(NM)$ if it is sparse.

Compared to matrix decomposition, low-rank CP decomposition is unique under mild conditions with the exception of "the elementary indeterminacies of permutation and scaling" [89]. Matrix decomposition is never essentially unique, unless the rank is one or imposing additional constraints on low-rank factors [89]. However, given a particular tensor, determining its rank is NP-hard [89]. Therefore, the rank is often estimated by trial and error or by assumptions made on the data. A conceptual limitation of CPD is that, if we attempt to decompose a tensor as a set of relationships between "concepts," this representation assumes that every tensor mode is involved with every other concept. If our data is noisy and of high order, this assumption may be overly strong and may introduce artificial relationships.

Despite these limitations, CPD has been proven both scalable and effective in broad applications [89, 150]. Take, as an example, the use of CPD in healthcare analytics. There, analysts have used CPD to identify useful medical concepts, or phenotypes, from raw electronic health records (EHR), which can then be used by medical professionals to facilitate diagnosis and treatment [75, 175]. Given a target rank R, CPD provides the top-R patient phenotypes, where each rank-one component of the decomposition corresponds to a phenotype.

The most popular algorithm to compute a CPD is arguably the alternating least squares method, or CP-ALS [37, 89]. Each iteration of CP-ALS loops over all modes and updates each mode's factor matrix while keeping the other factors constant. This process repeats

until user-specified conditions for convergence or maximum iteration counts are met. An N^{th}-order CP-ALS algorithm appears in algorithm 1. Line 5 is the mode-n MTTKRP. The combined result of all mode-$\{1, \ldots, N\}$ MTTKRPs is called the N^{th}-order MTTKRP *sequence* which will be referred to in Chapter 3.

Algorithm 1 The Nth-order CP-ALS algorithm.

Input: An Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and an integer rank R;
Output: Dense factors $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$, $\mathbf{U}^{(n)} \in R^{I_n \times R}$ and weights $\boldsymbol{\lambda}$;
1: Initialize $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$;
2: **do**
3: **for** $n = 1, \ldots, N$ **do**
4: $\mathbf{V} \leftarrow \mathbf{U}^{(1)\dagger}\mathbf{U}^{(1)} * \cdots \mathbf{U}^{(n-1)\dagger}\mathbf{U}^{(n-1)} * \mathbf{U}^{(n+1)\dagger}\mathbf{U}^{(n+1)} * \cdots * \mathbf{U}^{(N)\dagger}\mathbf{U}^{(N)}$
5: $\tilde{\mathbf{U}}^{(n)} \leftarrow \mathbf{X}_{(n)}(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(n+1)} \odot \mathbf{U}^{(n-1)} \odot \cdots \odot \mathbf{U}^{(1)})$
6: $\mathbf{U}^{(n)} \leftarrow \tilde{\mathbf{U}}^{(n)}\mathbf{V}^{\dagger}$
7: Normalize columns of $\mathbf{U}^{(n)}$ and store the norms as $\boldsymbol{\lambda}$
8: **while** Fit ceases to improve or maximum iterations exhausted.
9: **Return:** $[\![\boldsymbol{\lambda}, \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$;

2.3.2 Tucker Decomposition

The Tucker Decomposition, introduced in 1966 [165], produces a dense core tensor $\mathcal{G}$ multiplied by a factor matrix in every mode (figure 2.6(b)). Tucker expands algebraically as follows:

$$\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{U}^{(1)} \cdots \times_N \mathbf{U}^{(N)} \equiv [\![\mathcal{G}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!], \tag{2.7}$$

where $\mathcal{G} \in \mathbb{R}^{R_1 \times \cdots \times R_n}$ and $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R_n}$. $R_n, n = 1, \ldots, N$, named the n-rank of $\mathcal{X}$, is the column rank of $\mathbf{X}_{(n)}$. The tuple $(R_1, R_2, \ldots, R_N)$ is called the *multilinear rank* of Tucker decomposition. Different from the canonical rank used in CPD, multilinear ranks are relatively easy to determine. Therefore, we can find an exact Tucker decomposition of rank $(R_1, R_2, \ldots, R_N)$ for a given tensor. However, the Tucker decomposition is not unique. The core tensor can be modified without affecting the fit as long as we apply the inverse modification to the factor matrices. This freedom opens the door to choosing transformations that simplify the core structure in some way so that most of its elements become zeros, thereby eliminating interactions between corresponding components and improving

uniqueness. The format $[\![\mathcal{G}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$ is called a Tucker tensor [90], which is used to interpret or compress tensor $\mathcal{X}$. In fact, CPD can be viewed as a special case of Tucker decomposition, with diagonal core tensor $\mathcal{G}$ constituted by entries $\{\lambda_1, \ldots, \lambda_N\}$.

For a hypercubical tensor $\mathcal{X} \in \mathbb{R}^{I \times \cdots \times I}$ and a unified rank R, a Tucker tensor has $\mathcal{O}\big(R^N + NIR\big)$ parameters, which is smaller than $\mathcal{O}\big(I^N\big)$ if it is dense or $\mathcal{O}(NM)$ if it is sparse but larger than a Kruskal tensor from CPD. If the input data is high-dimensional but sparse, which is the case in many data applications, the storage of the dense core tensor may become problematic as it grows exponentially in the number of modes.

The Higher-Order Orthogonal Iteration (HOOI) algorithm is a popular Tucker decomposition algorithm [53]. In each of its iterations, the algorithm updates all factor matrices at once. In this dissertation, we study HOOI based on the alternating least squares paradigm (Algorithm 2). This HOOI algorithm takes random factor matrices or a decomposition produced by Higher-Order SVD (HOSVD) [52] as the initial values and then iterates until a predefined convergence rate is satisfied or some maximum number of iterations has occurred. Within an iteration, each factor matrix is updated by a TTM-chain and an SVD operation in its corresponding mode. After all factor matrices have converged, the core tensor $\mathcal{G}$ is computed. Therefore, TTM-chain and SVD are the dominant computations of the Tucker decomposition.

Algorithm 2 Tucker decomposition algorithm (HOOI) with ALS.

Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, and a set of ranks $R_1, \ldots, R_N$;
Output: Factor matrices $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$, $\mathbf{U}^{(n)} \in R^{I_n \times R}$ and a core tensor $\mathcal{G}$;
1: Initialize $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$ randomly or using HOSVD
2: **do**
3: **for** $n = 1, \ldots, N$ **do**
4: $\mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{U}^{(1)} \cdots \times_{(n-1)} \mathbf{U}^{(n-1)} \times_{(n+1)} \mathbf{U}^{(n+1)} \cdots \times_N \mathbf{U}^{(N)}$
5: $\mathbf{U}_n \leftarrow R_n$ leading left singular vectors of $\mathcal{Y}_{(n)}$
6: **while** fit ceases to improve or maximum iterations exhausted
7: $\mathcal{G} \leftarrow \mathcal{X} \times_1 \mathbf{U}^{(1)} \cdots \times_N \mathbf{U}^{(N)}$
8: **Return** $\mathcal{G}, \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$;

Although there are other higher-order decompositions, the above two are the fundamental ones and contain computational patterns representative of other variations. In applica-

tions, CP and Tucker decompositions have their own advantages and disadvantages. Given a tensor with known low-rank property, CP decomposition could be enough to compute quickly; otherwise, Tucker decomposition may be needed to compute a more accurate representation. Notice that the factor matrices of CP and Tucker decompositions are dense in general, which makes *semi-sparse* tensor objects useful during computation, as we discuss in Chapter 7.

2.4 Parallel Computer Architectures

Ultimately, we are interested in implementations on modern parallel computing platforms. The specific classes of systems we target are summarized below.

2.4.1 Multicore CPUs

Multicore CPUs become the mainstream in various devices, such as personal computers (PCs), tablets, even cell phones, to name a few. Take the example of a four-socket Intel Xeon E7-4850 v3 machine consisting 56 physical cores, each with 2.2 GHz frequency. It is a Haswell platform with 32 KiB L1 data cache and 1970 GiB memory. Its peak memory bandwidth is 273 GB/s. Because of its tens of threads and non-uniform memory access (NUMA) memory design, reasonable thread scalability is important to a parallel algorithm's performance.

2.4.2 NVIDIA GPU Architecture

NVIDIA Graphic Processor Units (GPUs) are highly parallel, many-core processors with high memory bandwidth [1]. We particularly consider GPUs that support CUDA, a general-purpose parallel computing platform and programming model. A GPU consists of many CUDA cores, which are organized as groups of Streaming Multiprocessors (SMs). The P100 GPU card used in this work employs Pascal architecture, which integrates 56 SMs and each SM consists of 64 CUDA cores. All cores in an SM share 64KB configurable on-chip

memory, L1 cache and shared memory. Three configurations are available: 16KB/48KB, 32KB/32KB, and 48KB/16KB for L1 cache and shared memory respectively. Global memory is cached by both L1 and L2 caches depending on the data is writable or read-only. Writable global memory data can only be cached by the L2 cache, read-only global memory data also can be cached by the L1 cache by marking the data with both "`const`" and "`__restrict__`" keywords. For performance reasons, it is necessary to carefully coordinate memory access paths for different types of data. CUDA has three-level thread hierarchy, grids, thread blocks, and threads. The maximum of their sizes varies among different GPU architectures. A P100 GPU supports up to 1024 threads per thread block. Threads are executed simultaneously in warps, usually a group of 32 threads. It is critical to have enough blocks and threads, good thread behavior inside a warp, and efficient memory access to achieve a satisfiable performance of CUDA programs.

2.4.3 Intel Xeon Phi Processors

Intel Xeon Phi processors are manycore architectures that delivers massive parallelism and vectorization. Knights Landing (KNL) is the codename for the second generation Xeon Phi, we use Intel Xeon Phi Processor 7250 as an example. It has 68 physical cores, each has 1.4 GHz frequency four-way simultaneous multi-threading and two 512-bit vector processing units. Each core has 32KB L1 cache and two cores share 1MB L2 cache. KNL supports differently configuring multi-channel DRAM (MCDRAM) memory to three modes: flat mode (explicitly managed by software), cache mode (as a last-level cache for DDR4), or hybrid mode. MCDRAM offers up to 490 GB/s memory bandwidth in flat mode, measured by the STREAM benchmark [114], which is approximately $2\times$ the bandwidth of 4-socket Xeon E7-4850.

CHAPTER 3

ADATM: MEMOIZED SPARSE CPD

This chapter concerns performance enhancement techniques for the CANDECOMP/PARAFAC decomposition (CPD) for sparse tensors. The running time of a typical CPD on an Nth-order tensor is dominated by the evaluation of a sequence of N *matricized tensor times Khatri-Rao product* (MTTKRP) operations (§ 3.1) [41, 80, 85, 156]. Prior performance studies have focused on optimizing a single MTTKRP [41, 80, 85, 156]. By contrast, we look for ways to improve the *entire* sequence of N MTTKRP*s*, which can lead to a much faster implementation, especially when N is large. A similar idea has been proposed by Phan et al. [137]; however, our method applies to the sparse (rather than dense) case and exploits the structure of an MTTKRP sequence to preserve sparsity and thereby reduce space.

Here, we propose a novel, adaptive tensor memoization algorithm, which we refer to as ADATM.[1] To perform a rank-R CPD of an Nth-order sparse tensor with m non-zeros, prior implementations require $\mathcal{O}\big(N^{(1+\epsilon)}mR\big)$ floating-point operations (flops), where $\epsilon \in [0, 1]$ is an implementation- and input-dependent parameter [15, 41, 80, 85, 156]. By contrast, our proposed method reduces this flop-complexity to $\mathcal{O}\Big(\tilde{N}mR\Big)$, where $\tilde{N}$ is usually much less than $N^{(1+\epsilon)}$; furthermore, the user may control the degree of improvement by trading increased storage for reduced time (smaller $\tilde{N}$) (§ 3.3).

Our method has several parameters, such as tensor features (order, size, non-zero distribution), target rank, and memory capacity. Thus, we develop a model-driven framework to tune them. By model-driven, we mean the framework includes a predictive model for pruning the space candidate implementations, using a user-selectable strategy to prioritize time or space concerns. We further accelerate ADATM within a node by multithreading

[1] Read, "Ada-Tee-Em," as in Ada Lovelace. This work has been published [101].

(§ 3.4).

ADATM's MTTKRP sequence outperforms the state-of-the-art SPLATT [157] and Tensor Toolbox [16] by up to $8\times$ and $820\times$, respectively, on real sparse tensors with orders as high as 85. Our predictive model effectively selects an optimal implementation. Compared to previous work [41, 80, 85, 156], ADATM scales better as the order (N) grows. For CPD, ADATM achieves up to $8\times$ speedups over SPLATT (§ 3.5).

Table 3.1 summarizes the symbols and notations will be used only in Chapter 3, other general symbols are listed in table 2.1.

Table 3.1: List of symbols and notations in Chapter 3.

Symbols	Description
Operations	
$\mathcal{X} \diamond_n \mathbf{U}$	Mode-n quasi tensor-times-matrix product (q-TTM)
Input parameters	
M_l	#Nodes at level l of a CSF tree of tensor $\mathcal{X}$
S_{mem}	Machine memory size
CSF Data structures	
csfinds	Indices of CSF, β_{int} bits
ptrs	Pointers of CSF, β_{int} or β_{long} bits
ADATM parameters	
T_{CP}	Time of CP-ALS
T_{MTTKRP}	Time of a single MTTKRP
n_p	#Memoized MTTKRPs in an MTTKRP sequence (#Producer modes)
n_c	#Non-memoized MTTKRPs in an MTTKRP sequence (#Consumer modes)
n_O	#Tensor products (TTM or q-TTM) in an MTTKRP sequence
n_i	#Saved intermediate tensors from a memoized MTTKRP
s	Predicted storage size of ADATM
t	Predicted running time of ADATM

3.1 Why to Optimize the MTTKRP Sequence?

3.1.1 MTTKRP is the performance bottleneck of CP-ALS.

Consider an Nth-order sparse hypercubical tensor $\mathcal{X} \in R^{I \times \cdots \times I}$ with M non-zeros. The number of flops in *one* iteration of CP-ALS (algorithm 1) is approximately

$$T_{CP} \approx N(N^{\epsilon}MR + NIR^2) \approx NT_{MTTKRP}, \tag{3.1}$$

where $T_{MTTKRP} = \mathcal{O}(N^{\epsilon}MR)$ is the time for a single MTTKRP [16, 41, 85, 156]. (The NIR^2 term will typically be neglible because in practice $IR \ll M$.) Tensor Toolbox [16, 85] has $\epsilon = 1$, while SPLATT [156] and DFacTo [41] have $\epsilon \approx 0$ for third-order sparse tensors and $\epsilon \in (0, 1)$ for higher-order sparse tensors.[2] The value of ϵ depends on both the implementation and sparse tensor features, e.g., mode sizes, non-zero distribution. MTTKRPs dominate the running time of CP-ALS: We ran CP-ALS using the state-of-the-art SPLATT library [156] on all of the tensors in table 3.4 (see § 3.5) and found that MTTKRP accounted for 69-99% of the total running time. Therefore, we focus on optimizing the sequence of MTTKRPs.

3.1.2 The time of an MTTKRP sequence grows with tensor order.

Researchers have successfully optimized a single MTTKRP operation through various methods [41, 80, 85, 156], which reduces the hidden constant of T_{MTTKRP}. However, the overall time T_{CP} still grows with the tensor order since CP-ALS executes a *sequence* of N MTTKRPs.

Figure 3.1 shows the runtime of an MTTKRP sequence using SPLATT on synthetic, hypercubical, sparse tensors generated from Tensor Toolbox [16]. The tensor orders increase from 10 to 80, while the mode size (1,000), rank size R (16) and number of non-zeros

[2] SPLATT and DFacTo improve T_{MTTKRP} of a third-order sparse tensor from $3MR$ to $2(M + P)R \approx \mathcal{O}(MR)$ where P is the number of non-empty mode-n fibers, $P \ll M$. However, this $P \ll M$ does not apply when N is large, thus $\epsilon \in (0, 1)$ instead.

(100,000) remain fixed. These synthetic tensors have $\epsilon = 1$, thus the runtime of an MTTKRP sequence increases close to quadratically with N, since each MTTKRP grows with N linearly per equation (3.1) and the sequence performs N MTTKRPs. Others have improved an MTTKRP sequence by saving large Khatri-Rao product results [137, 190]. In this chapter, we propose a new time and space efficient algorithm to solve this problem.

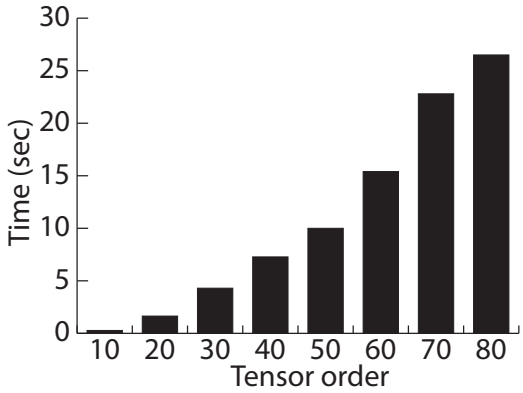


Figure 3.1: The runtime of SPLATT sequence on synthetic, sparse tensors.

3.1.3 An MTTKRP sequence has arithmetic redundancy.

In the Nth-order MTTKRP sequence in algorithm 1, each mode-n MTTKRP shares all but one factor matrix with the mode-$(n-1)$ MTTKRP. Figure 3.2 gives an example of factoring a fourth-order tensor, the mode-1 MTTKRP $\tilde{\mathbf{A}} \leftarrow \mathcal{X}_{(1)}(\mathbf{D} \odot \mathbf{C} \odot \mathbf{B})$ and the mode-2 MTTKRP $\tilde{\mathbf{B}} \leftarrow \mathcal{X}_{(2)}(\mathbf{D} \odot \mathbf{C} \odot \tilde{\mathbf{A}})$ redundantly compute $\mathbf{D}$ and $\mathbf{C}$ with tensor $\mathcal{X}$. Same redundancy happens between mode-2 and mode-3, mode-3 and mode-4 MTTKRPs. Theoretically, if we could save all of the intermediate results from the mode-1 MTTKRP, we could avoid about $\frac{1}{2}N^2$ redundant computations. The number of redundant computations quadratically increases with the tensor order, which is not well scalable. Our proposed method seeks to memoize these redundant computations.

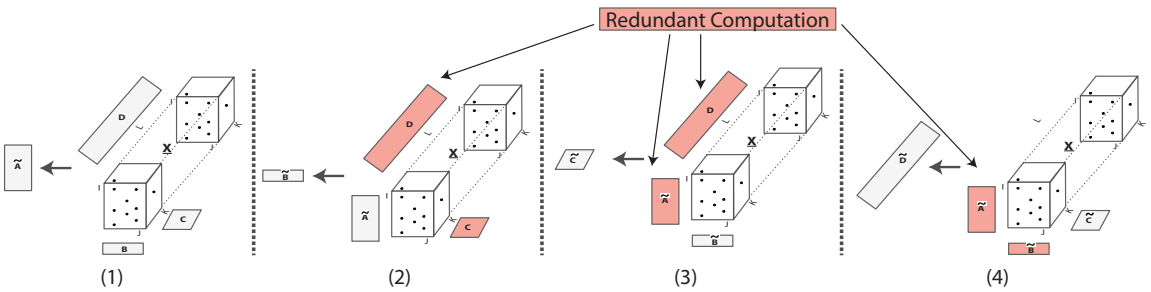


Figure 3.2: A fourth-MTTKRP sequence of a CPD.

3.2 MTTKRP Algorithms and Semi-sparse Tensor Format (vCSF)

We introduce two standalone MTTKRP algorithms, which are the building blocks for our memoization scheme. We also propose vCSF format for semi-sparse tensors, which is an auxiliary format of CSF.

3.2.1 MTTKRP Algorithms

We first introduce a simplifying building block, ***quasi tensor times matrix multiplication* (q-TTM)** through Hadamard product. Given an $(N+1)$th-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N \times R}$ and a matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, the q-TTM product in mode n is denoted by $\mathcal{Y} = \mathcal{X} \diamond_n \mathbf{U}$, where $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N \times R}$ is an Nth-order tensor. In scalar form, q-TTM is

$$y(i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N, r) = \sum_{i_n=1}^{I_n} x(i_1, \ldots, i_{n-1}, i_n, i_{n+1}, \ldots, i_N, r) u(i_n, r). \quad (3.2)$$

By fixing indices $i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N$, a Hadamard product is taken between each slice $\mathbf{X}(i_1, \ldots, i_{n-1}, :, i_{n+1}, \ldots, i_N, :)$ and the matrix $\mathbf{U}$. Every resulting $I_n \times R$ matrix is sum-reduced along with mode-n for all i_ns.

MTTKRP could operate on the non-zeros of a sparse tensor $\mathcal{X}$ without explicitly matricizing. It also avoids explicit Khatri-Rao products, thereby saving space. For example,

consider the mode-1 MTTKRP of the fourth-order example:

$$\tilde{\mathbf{A}} = \mathbf{X}_{(1)}(\mathbf{D} \odot \mathbf{C} \odot \mathbf{B}), \tag{3.3}$$

Smith et al. proposed an MTTKRP algorithm, SPLATT [156], which factors out the inner multiplication with $\mathbf{B}$ and $\mathbf{C}$ thereby reducing the number of flops:

$$\tilde{A}(i,r) = \sum_{j=1}^{J} B(j,r) \sum_{k=1}^{K} C(k,r) \sum_{l=1}^{L} \mathcal{X}(i,j,k,l) D(l,r). \tag{3.4}$$

We define two types of standalone MTTKRP algorithms to help clarify our memoization approach. A *memoized* MTTKRP is the traditional MTTKRP computed from scratch using SPLATT [156] and saving its intermediate results, the semi-sparse tensors (algorithm 3). A *partial* MTTKRP is the MTTKRP computed based on the saved intermediate results from a memoized MTTKRP. It generally takes less time than a memoized MTTKRP. (See its embedded usage in algorithm 4 and 5 in § 3.3.)

Algorithm 3 starts with a TTM, yielding a semi-sparse tensor $\mathcal{Y}^{(1)} \in \mathbb{R}^{I \times J \times K \times R}$, where mode-4 is dense. Then, a q-TTM is performed on $\mathcal{Y}^{(1)}$ with $\mathbf{C}$ to generate the second semi-sparse tensor, $\mathcal{Y}^{(2)}$, with a dense mode-3; then a second q-TTM is performed on $\mathcal{Y}^{(2)}$ with $\mathbf{B}$ to produce the final result $\tilde{\mathbf{A}}$. In this chapter, we consider an MTTKRP as the integration of the two *products* TTM and q-TTM. For an arbitrary Nth-order tensor, a memoized MTTKRP has $\mathcal{O}(N^{\epsilon}MR)\,, \epsilon \in [0,1)$ flops, while the number of products is $(N-1)$.

Algorithm 3 Memoized MTTKRP algorithm using SPLATT.

Input: A fourth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K \times L}$, dense factors $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$, $\mathbf{D} \in \mathbb{R}^{L \times R}$;
Output: Updated dense factor matrix $\tilde{\mathbf{A}} \in \mathbb{R}^{I \times R}$;

▷ $\tilde{\mathbf{A}} \leftarrow \mathcal{X}_{(1)}(\mathbf{D} \odot \mathbf{C} \odot \mathbf{B})$

1: Save $\mathcal{Y}^{(1)}$: $\mathcal{Y}^{(1)} \leftarrow \mathcal{X} \times_4 \mathbf{D}$
2: Save $\mathcal{Y}^{(2)}$: $\mathcal{Y}^{(2)} = \mathcal{Y}^{(1)} \diamond_3 \mathbf{C}$
3: $\tilde{\mathbf{A}} = \mathcal{Y}^{(2)} \diamond_2 \mathbf{B}$
4: **return** $\tilde{\mathbf{A}}$;

3.2.2 Sparse Tensor Product Property

Property SPTTM *outputs a semi-sparse tensor whose product mode is dense, while index modes remain unchanged.*

Proof. Assume an SPTTM takes a sparse tensor $\mathcal{X}$, a dense matrix $\mathbf{U}$ as inputs and a tensor $\mathcal{Y}$ as output. Mode-n fibers of $\mathcal{X}$ and $\mathcal{Y}$ tensors are defined as

$$\mathbf{f}_n^X = \mathcal{X}(i_1, \ldots, i_{n-1}, :, i_{n+1}, \ldots, i_N),$$

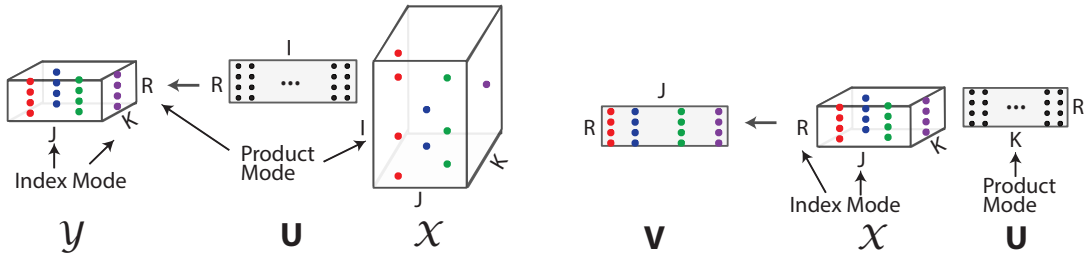
$$\mathbf{f}_n^Y = \mathcal{Y}(i_1, \ldots, i_{n-1}, :, i_{n+1}, \ldots, i_N),$$

where $i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N$ are given. $\mathbf{f}_n^X$ is a sparse fiber because of the sparsity of $\mathcal{X}$. Thus, Equation 2.2 is equal to

$$f_n^Y(r) = \sum_{i_n=1}^{I_n} f_n^X(i_n) u_{i_n r}. \tag{3.5}$$

When r is fixed, element $\mathbf{f}_n^Y(r)$ is a dot-product of fiber $\mathbf{f}_n^X$ and $\mathbf{u}(:, r)$, a column of $\mathbf{U}$. Since $\mathbf{u}(:, r)$ is a dense vector, each $f_n^Y(r)$ is non-zero if at least one non-zero exists in fiber $\mathbf{f}_n^X$. That is, a non-empty fiber $\mathbf{f}_n^X$ generates a dense fiber $\mathbf{f}_n^Y$. For each pair of fixed indices $(i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N)$, we compute a mode-n fiber of $\mathcal{Y}$, which shows the indices $i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N$ are unchanged for the resulting tensor $\mathcal{Y}$. Figure 3.3(a) shows the behavior of a third-order sparse tensor times a dense matrix. The product mode 1 (size-I) is a dense mode in the resulting tensor $\mathcal{Y}$ (size-R), while its index mode $2, 3$ (size-J, K) is the same with the input sparse tensor, except it indexes dense fibers of the output. □

Similarly, the q-TTM of a semi-sparse tensor and a dense matrix yields another semi-sparse tensor whose index modes are unchanged while its product mode(s) disappears. Figure 3.3(b) shows the behavior of a third-order semi-sparse tensor times a dense matrix. The product mode 3 (size-K) does not exist in the resulting matrix $\mathbf{V}$, while its index mode



(a) Sparse tensor-dense matrix multiply (b) Semi-sparse tensor-dense matrix multiply

Figure 3.3: Sparse tensor product property in third-order tensors.

$1, 2$ (size-R, J) is the same with the input sparse tensor.

3.2.3 vCSF Format

Besides the algorithms, we need to specify a data structure for the sparse tensors. More recently, Smith et al. proposed a Compressed Sparse Fiber (CSF) format (figure 3.4(b)) [152] which is memory-efficient and generally leads to a faster MTTKRP compared to COO format [152, 156]. Therefore, our tensor memoization algorithm assumes CSF for storing the input sparse tensor; to store the intermediate semi-sparse tensors, we use a variant of CSF, which we call vCSF, shown in figure 3.4(c).

In CSF, each root-to-leaf path corresponds with a coordinate tuple of a non-zero entry, where replicated indices at the same level are compressed to reduce storage. When computing a sparse tensor-dense matrix product (TTM), the output tensor $\mathcal{Y}^{(1)}$ in algorithm 3 has the same i, j, k indices with $\mathcal{X}$ and a dense mode-l (see § 3.2.2). Thus, when storing a semi-sparse tensor $\mathcal{Y}^{(1)}$, we do not need to store indices of all modes since i, j, k can be reused from $\mathcal{X}$ and dense indices l need not to be stored. We refer to this storage scheme as vCSF, as an auxiliary format of CSF, storing only the non-zero values of an intermediate semi-sparse tensor.

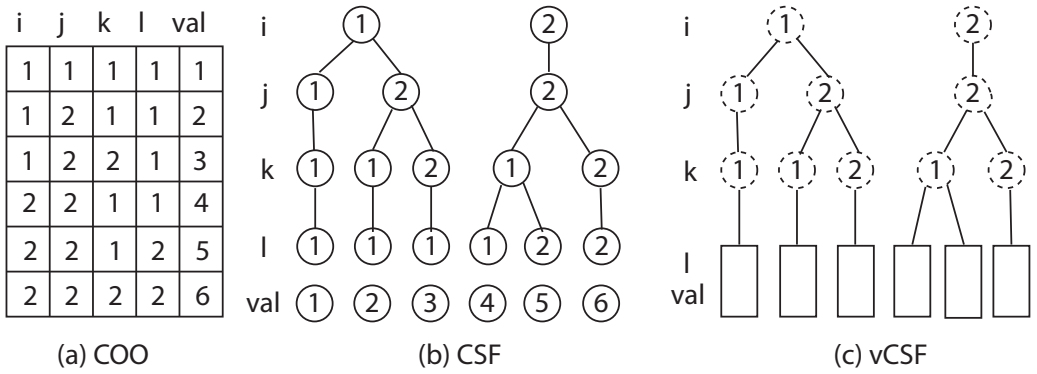


Figure 3.4: A sparse tensor $\mathcal{X} \in \mathbb{R}^{2\times2\times2\times2}$ in COO and CSF formats, tensor $\mathcal{Y}^{(1)}$ in vCSF format without storing indices.

3.3 Memoization for the MTTKRP Sequence

We present two tensor memoization algorithms for the MTTKRP sequence. To simplify the presentation, we show them assuming the fourth-order example but have implemented the general algorithm for Nth-order tensors.

3.3.1 Two Fourth-Order Tensor Memoization Algorithms

A Simple Memoization Scheme

A simple memoized algorithm appears in algorithm 4, shown for simplicity for the fourth-order case ($N = 4$). It saves every intermediate semi-sparse tensor ($\mathcal{Y}^{(1)}$ and $\mathcal{Y}^{(2)}$) generated from the mode-1 MTTKRP using the memoized MTTKRP algorithm (algorithm 3) and then reuses them to speedup subsequent MTTKRPs using the partial MTTKRP algorithm. Note that the mode-3 MTTKRP has more flops than the mode-2 one. The mode-4 MTTKRP cannot reuse any saved intermediates, therefore, it is computed from scratch using SPLATT [156], which directly operates on the tensor $\mathcal{X}$ and does not save any intermediate tensors. Figure 3.5 graphically illustrates this algorithm.

Algorithm 4 A simple tensor memoization algorithm of a fourth-order sparse MTTKRP sequence.

Input: A fourth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K \times L}$, dense initial factors $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$, $\mathbf{D} \in \mathbb{R}^{L \times R}$;
Output: Updated factors $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}}, \tilde{\mathbf{D}}$;

1: Save $\mathcal{Y}^{(1)}$: $\mathcal{Y}^{(1)} \leftarrow \mathcal{X} \times_4 \mathbf{D}$; ▷ $\tilde{\mathbf{A}} \leftarrow \mathcal{X}_{(1)}(\mathbf{D} \odot \mathbf{C} \odot \mathbf{B})$.
2: Save $\mathcal{Y}^{(2)}$: $\mathcal{Y}^{(2)} \leftarrow \mathcal{Y}^{(1)} \diamond_3 \mathbf{C}$;
3: $\tilde{\mathbf{A}} \leftarrow \mathcal{Y}^{(2)} \diamond_2 \mathbf{B}$; ▷ $\tilde{\mathbf{B}} \leftarrow \mathcal{X}_{(2)}(\mathbf{D} \odot \mathbf{C} \odot \tilde{\mathbf{A}})$.
4: Reuse $\mathcal{Y}^{(2)}$: $\tilde{\mathbf{B}} \leftarrow \mathcal{Y}^{(2)} \diamond_1 \tilde{\mathbf{A}}$ ▷ $\tilde{\mathbf{C}} \leftarrow \mathcal{X}_{(3)}(\mathbf{D} \odot \tilde{\mathbf{B}} \odot \tilde{\mathbf{A}})$.
5: Reuse $\mathcal{Y}^{(1)}$: $\tilde{\mathbf{C}} \leftarrow (\mathcal{Y}^{(1)} \diamond_2 \tilde{\mathbf{B}}) \diamond_1 \tilde{\mathbf{A}}$ ▷ $\tilde{\mathbf{D}} \leftarrow \mathcal{X}_{(4)}(\tilde{\mathbf{C}} \odot \tilde{\mathbf{B}} \odot \tilde{\mathbf{A}})$.
6: SPLATT: $\tilde{\mathbf{D}} \leftarrow \mathcal{X}_{(4)}(\tilde{\mathbf{C}} \odot \tilde{\mathbf{B}} \odot \tilde{\mathbf{A}})$
7: **Return:** $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}}, \tilde{\mathbf{D}}$;

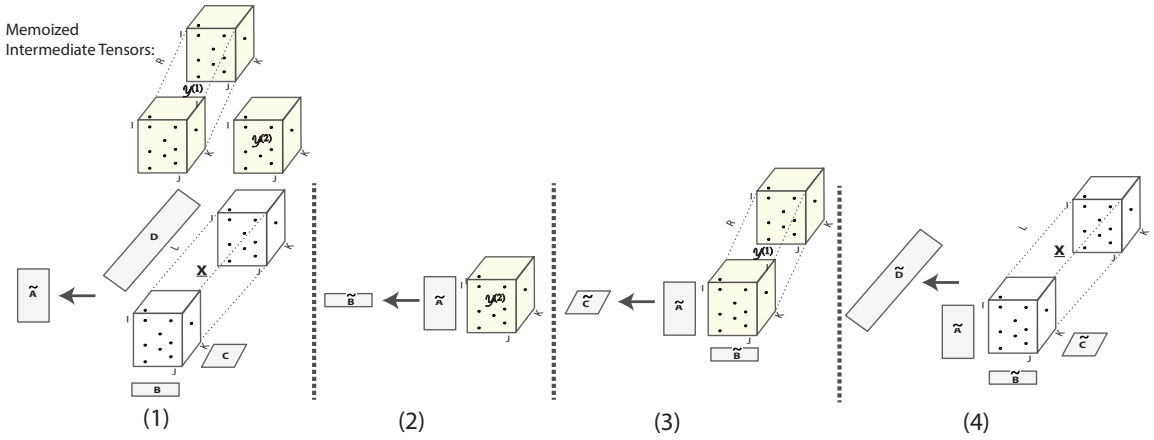


Figure 3.5: Graphical description of the simple tensor memoization algorithm.

An Optimal Memoized Scheme

The simple method of algorithm 4, which memoizes all intermediates from one MTTKRP in a sequence, is not the only or the best approach.

For instance, consider algorithm 5. Whereas algorithm 4 memoizes only the mode-1 MTTKRP, algorithm 5 memoizes with respect to the mode-1 and mode-3 MTTKRPs. In particular, it saves only one intermediate tensor $\mathcal{Y}^{(2)}$ during the mode-1 MTTKRP, which is then reused in the mode-2 MTTKRP; then, during the mode-3 MTTKRP it saves $\mathcal{Z}^{(2)}$. To be able to apply $\mathcal{Z}^{(2)}$ during the mode-4 MTTKRP, we need to permute $\mathcal{X}$, which in

our scheme creates another sparse tensor $\tilde{\mathcal{X}} \in \mathbb{R}^{K \times J \times I \times L}$. (It is possible to avoid this extra storage cost but we do not consider that in this work to pursue better performance.) Figure 3.6 graphically illustrates this algorithm.

Algorithm 5 An optimal tensor memoization algorithm of a fourth-order sparse MTTKRP sequence.

Input: A fourth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K \times L}$, dense initial factors $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$, $\mathbf{D} \in \mathbb{R}^{L \times R}$;
Output: Updated factors $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}}, \tilde{\mathbf{D}}$;

$\triangleright\ \tilde{\mathbf{A}} \leftarrow \mathcal{X}_{(1)}(\mathbf{D} \odot \mathbf{C} \odot \mathbf{B})$.

1: Save $\mathcal{Y}^{(2)}$: $\mathcal{Y}^{(2)} \leftarrow (\mathcal{X} \times_4 \mathbf{D}) \diamond_3 \mathbf{C}$;
2: $\tilde{\mathbf{A}} \leftarrow \mathcal{Y}^{(2)} \diamond_2 \mathbf{B}$;

$\triangleright\ \tilde{\mathbf{B}} \leftarrow \mathcal{X}_{(2)}(\mathbf{D} \odot \mathbf{C} \odot \tilde{\mathbf{A}})$.

3: Reuse $\mathcal{Y}^{(2)}$: $\tilde{\mathbf{B}} \leftarrow \mathcal{Y}^{(2)} \diamond_1 \tilde{\mathbf{A}}$

$\triangleright\ \tilde{\mathbf{C}} \leftarrow \mathcal{X}_{(3)}(\tilde{\mathbf{B}} \odot \tilde{\mathbf{A}} \odot \mathbf{D})$.

// Permute $\mathcal{X}$ to $\tilde{\mathcal{X}} \in \mathbb{R}^{K \times L \times I \times J}$.
4: Save $\mathcal{Z}^{(2)}$: $\mathcal{Z}^{(2)} \leftarrow (\tilde{\mathcal{X}} \times_4 \tilde{\mathbf{B}}) \diamond_3 \tilde{\mathbf{A}}$;
5: $\tilde{\mathbf{C}} \leftarrow \mathcal{Z}^{(2)} \diamond_2 \mathbf{D}$;

$\triangleright\ \tilde{\mathbf{D}} \leftarrow \mathcal{X}_{(4)}(\tilde{\mathbf{B}} \odot \tilde{\mathbf{A}} \odot \tilde{\mathbf{C}})$.

6: Reuse $\mathcal{Z}^{(2)}$: $\tilde{\mathbf{D}} \leftarrow \mathcal{Z}^{(2)} \diamond_1 \tilde{\mathbf{C}}$
7: **Return:** $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}}, \tilde{\mathbf{D}}$;

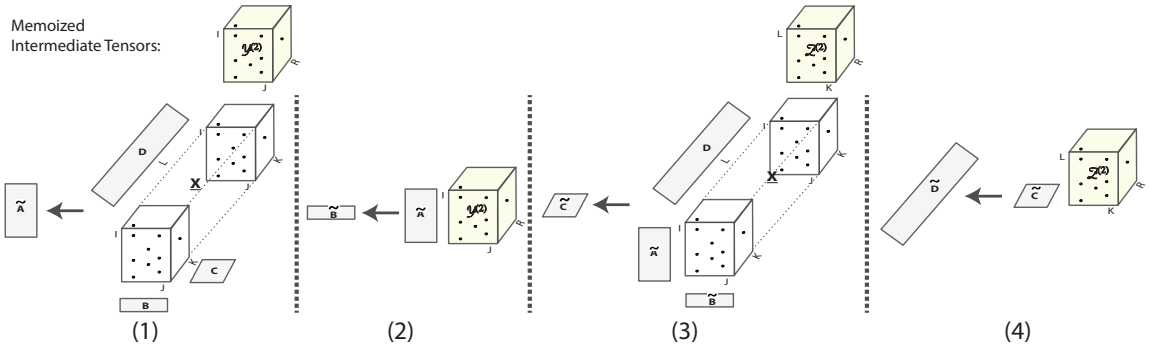


Figure 3.6: Graphical description of the optimal tensor memoization algorithm.

For better clarification, we summarize the three algorithms (scratch (SPLATT), simple and optimal tensor memoization algorithms) in figure 3.7. The optimal algorithm needs 8 products counting both TTM and q-TTM products, which is one less than the simple algorithm, though the permuted sparse tensor $\tilde{\mathcal{X}}$ takes some extra space.

For an Nth-order tensor, the simple and optimal algorithms (Algorithm 4 and 5) memoize 1 and $n_p \in [1, \frac{N}{2}]$ MTTKRPs respectively, where n_p will be determined from § 3.3.2. The

simple algorithm only memoizes the mode-1 MTTKRP, and its total number of products of the MTTKRP sequence grows like $\mathcal{O}(N^2)$; by contrast, the optimal algorithm memoizes more MTTKRPs, reducing the number of products to $\mathcal{O}(N^{1.5})$ (see § 3.3.2). That is, this reduction is *asymptotic* in N. As it happens, this choice of memoization strategies is optimal under certain conditions. However, it also requires extra space to store intermediate tensors and the permuted tensor(s). Thus, choosing a strategy may require trading space for time. This motivates our proposed scheme, which tries to find a time-optimal strategy guided by a user-specified storage limit.

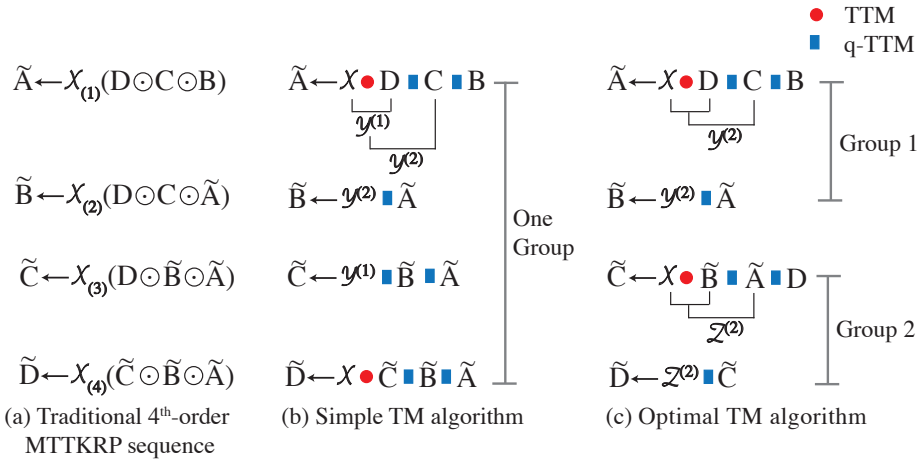


Figure 3.7: Graphical process of the simple and optimal tensor memoization algorithms. "red circle" represents TTM and "blue block" is q-TTM.

3.3.2 Memoization Strategy Analysis

To see how the choice of memoization strategy can lead to asymptotic reductions in operation count, consider a simplified analysis for the following problem:

> **Problem**: *Find the number of memoized* MTTKRPs *that minimizes the total number of products (*TTM *and q-*TTM*) in an Nth-order* MTTKRP *sequence.*

Though our experiments involve general sparse tensors, for this analysis suppose the input tensor $\mathcal{X} \in \mathbb{R}^{I \times \cdots \times I}$ is hypercubical and dense. Since each product may consume different amounts of flops, we assume an average number of flops per product that is fixed for a

given tensor $\mathcal{X}$. We also assume each partial MTTKRP reuses as many of the memoized intermediate tensors as possible.

A *producer mode* is the one that uses a memoized MTTKRP and produces intermediate tensors, while a *consumer mode* uses a partial MTTKRP that consumes these intermediates. One producer mode and some consumer modes constitute an MTTKRP *group*, in which all the consumer modes reuse the intermediate tensors that the producer mode memoized. Let the number of producer modes and consumer modes be n_p and n_c, respectively, where $n_c = N - n_p$; and let the total number of products in the Nth-order MTTKRP sequence be n_O. Thus, $n_p = 1$ and $n_O = \mathcal{O}(N^2)$ in algorithm 4 and $n_p = 2$ and $n_O = \mathcal{O}(N^{1.5})$ in algorithm 5.

Lemma 3.3.1. *Given n_p producer modes, the minimum n_O can be achieved by evenly assigning producer modes to consumer modes for reuse.*

Proof. Consider any assignment of c_i consumer modes to the i^{th} producer mode in group-i, where $i = 1, ..., n_p$. Then,

$$n_p + n_c = n_p + \sum_{i=1}^{n_p} c_i = N. \tag{3.6}$$

The products in group-i consists of $(N-1)$ products in the one memoized MTTKRP plus the products of c_i partial MTTKRPs. From figure 3.7(b), the number of products of all c_i consumer modes in group-i is $1 + \cdots + c_i = c_i(c_i+1)/2$; thus the number of products in group-i,

$$O_i = N - 1 + \frac{c_i(c_i+1)}{2}. \tag{3.7}$$

The total number of products of an MTTKRP sequence is

$$n_O = \sum_{i=1}^{n_p} O_i = \frac{1}{2}\left[\sum_{i=1}^{n_p} c_i^2 + \sum_{i=1}^{n_p} c_i + 2n_p(N-1)\right]. \tag{3.8}$$

Substituting equation (3.6) yields

$$n_O = \frac{1}{2}\left[\sum_{i=1}^{n_p} c_i^2 + 2n_pN + N - 3n_p\right]. \tag{3.9}$$

Since n_p and N are fixed, minimizing n_O is equivalent to minimizing the sum of squares, $\sum_{i=1}^{n_p} c_i^2$, subject to the constraint equation (3.6). This elementary optimization problem has a minimum when $c_1 = \cdots = c_{n_p}$, by applying the Cauchy-Schwarz inequality. Therefore, the n_p producer modes should be evenly assigned to approximately $\frac{n_c}{n_p}$ consumer modes for reuse in order to minimize n_O. □

Lemma 3.3.2. *$n_p^* = \sqrt{N/2}$ minimizes n_O for an Nth-order* MTTKRP *sequence.*

Proof. For simplicity, assume n_p divides N. Then, by applying lemma 3.3.1,

$$c_i = N/n_p - 1, i = 1, \ldots, n_p. \tag{3.10}$$

Substituting equation (3.10) into equation (3.9),

$$n_O = \frac{1}{2}\left[\frac{N^2}{n_p} + 2(N-1)n_p - N\right], 1 \leq n_p \leq \frac{N}{2}, \tag{3.11}$$

where the largest possible $\frac{N}{2}$ is choosing all n_p by memoizing every other mode.

By taking the derivative of n_O with respect to n_p, the minimum occurs at

$$n_p^* = \frac{N}{\sqrt{2(N-1)}} \approx \sqrt{\frac{N}{2}}. \tag{3.12}$$

The minimum n_O is $n_O^* \approx \sqrt{2}N^{1.5}$. □

We know $n_O = N(N+1)/2$ when $n_p = 1$ and $n_O = N^2/2$ when $n_p = N/2$. The optimal n_p^* achieves the minimum $n_O^* = \mathcal{O}(N^{1.5})$, which is asymptotically better especially for higher-order tensors. Lemma 3.3.2 shows that even if we have infinite storage space to memoize all $N/2$ MTTKRPs, that would not actually minimize the number of products. Intuitively, even though consumer modes benefit from the saved intermediate tensors, the memoized MTTKRP itself consumes $(N-1)$ operations. This analysis shows there is an optimal balance.

Given an input tensor, we first calculate the optimal number of memoized MTTKRPs based on these lemmas, and then design our adaptive tensor memoization algorithm (ADATM) such as algorithm 5 for fourth-order tensors. As we assumed, this optimal n_p is calculated

on dense, hypercubical tensors, it is not accurate for diverse sparse tensors. In § 3.4, we construct a model-driven framework to predict the optimal parameters considering input tensor features and storage efficiency.

3.4 ADATM: Adaptive Tensor Memoization

Based on the discussion of § 3.3, we are motivated to develop a framework to choose an optimal memoization strategy. This section explains our approach, which we refer to as the adaptive tensor memoization framework, ADATM.

3.4.1 Parameter Selection

The memoized algorithm has several natural parameters, which can be tuned to trade storage for time:

n_p The number of producer modes (or memoized MTTKRPs). Lemma 3.3.2 shows that, in theory, there exists an optimal choice for n_p; for hypercubical dense tensors, $n_p^* = \sqrt{N/2}$. Therefore, our adaptive framework heuristically considers all $n_p \in \{1, \ldots, \sqrt{N/2}\}$. (In our experiments, we find that the optimal n_p is always smaller than n_p^*.)

m_o The mode order of a sparse tensor, the same meaning with M_O in table 2.1 but not input-determined. As the discussion of algorithm 5 suggests, one may choose the order of modes for each memoized MTTKRP. The two memoized MTTKRPs in algorithm 5 are $\mathcal{X} \in \mathbb{R}^{I \times J \times K \times L}$ and $\tilde{\mathcal{X}} \in \mathbb{R}^{K \times L \times I \times J}$, with different mode orders in each case. The new $\tilde{\mathcal{X}} \in \mathbb{R}^{K \times L \times I \times J}$ is created that mode-4 MTTKRP is able to reuse the memoized semi-sparse tensor $\mathcal{Z}^{(2)} \in \mathbb{R}^{K \times L \times R}$. In fact, $\hat{\mathcal{X}} \in \mathbb{R}^{K \times L \times J \times I}$ can also generate the same $\mathcal{Z}^{(2)}$, so we heuristically choose to contract long modes first, as originally suggested in the SPLATT work [156]. Thus, if $J > I$, we prefer $\tilde{\mathcal{X}} \in \mathbb{R}^{K \times L \times I \times J}$.

n_i The number of intermediate semi-sparse tensors saved from a memoized MTTKRP. For every memoized MTTKRP, the choice of n_i allows us to trade space for time. The range of n_i is $\{1, \ldots, N/n_p - 1\}$.

For any choice of preceding parameters, we have a model that estimates the storage $s(n_p, m_o, n_i)$ in bytes and time $t(n_p, m_o, n_i)$ in flops (see below). This capability is what enables us to quickly search the parameter space. Besides, armed with this model, we are able to construct a model-driven framework to find good values of these parameters.

3.4.2 A Model-Driven Framework

Our model-driven framework tunes the parameters (n_p, m_o, n_i) and then selects an ADATM implementation, as illustrated in figure 3.8. The inputs are the order N of a sparse tensor, its number of non-zeros M, and the target rank R. The user may specify a memory limit and preferred strategy, e.g., maximize performance or minimize storage by a certain degree. The framework considers a large set of configurations, then uses a predictive model to estimate each configuration's storage and time and prune the candidates that, for instance, do not meet the memory limit.

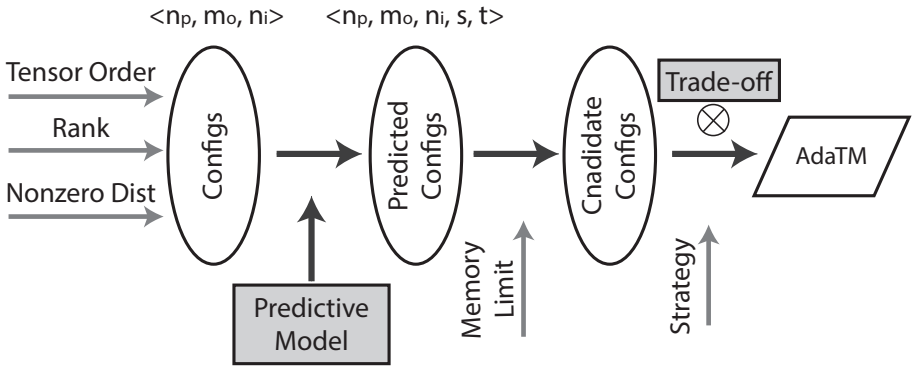


Figure 3.8: The model-driven framework of ADATM.

3.4.3 Predictive Model

The input sparse tensor is stored in the standard coordinate format, we predict the space s as the sum of n_p sparse tensors in CSF format and $n_p \times n_i$ semi-sparse tensors in vCSF format both under the mode order m_o. The time t is predicted by adding the flops of all products, TTMs and q-TTMs. The model for t need not be accurate in an absolute sense; rather, it need only be accurate enough to produce a correct relative ranking.

We show the formulas used for the estimates of s and t in table 3.2. The number of fibers at the l^{th}-level of a CSF tree is M_l for $l = 1, \ldots, N$ in figure 3.4(b), where $M_1 \leq \cdots \leq M_N = M$ (see [152] for details). We assume the sparse and semi-sparse tensors in evaluating TTM and q-TTM products are both in order-N and have M non-zeros. And since a CSF tensor is stored hierarchically, $M_{CSF} = 16 \sum_{l=1}^{N} M_l$. In q-TTM, the semi-sparse tensor in vCSF format only stores non-zero values, $8M$ bytes. One MTTKRP group consists of one memoized MTTKRP and $(N/n_p - 1)$ partial MTTKRPs. The memoized MTTKRP takes $2\sum_{l=2}^{N} M_l R = \mathcal{O}(N^{\epsilon} MR)$ time, $\epsilon \in [0, 1)$, since $M_l \leq M$, when $l < N$. The partial MTTKRPs in the same group, adhered to this memoized MTTKRP, have different non-zeros for each semi-sparse tensor. Finally, we sum all n_p MTTKRP groups up and get the time and space of ADATM algorithm for an MTTKRP sequence. When using SPLATT [156] to compute an MTTKRP sequence, the space is smaller than ADATM with the price of slower execution. From this table we get

$$s = \sum_{i=1}^{n_p} \left(M_{CSF}^{i} + 8 \sum_{l=\frac{N}{n_p}-n_i+1}^{\frac{N}{n_p}} M_l R \right) \tag{3.13}$$

$$t = 2 \sum_{i=1}^{n_p} \left(\sum_{l=2}^{N} M_l R + \sum_{l=1}^{\frac{N}{n_p}-1} \sum_{j=2}^{l+1} M_j \right) R \triangleq 2\tilde{N} MR. \tag{3.14}$$

Table 3.2: The number of flops and storage size of products and algorithms.

Algorithms		#Flops	Storage Size (Bytes)
Product	TTM	$2MR$	M_{CSF}
	q-TTM	$2MR$	$8M$
One MTTKRP group	Memoized MTTKRP	$2\sum_{l=2}^{N} M_l R$	$M_{CSF} + 8\sum_{l=\frac{N}{n_p}-n_i+1}^{\frac{N}{n_p}} M_l R$
	Partial MTTKRPs	$2\sum_{l=1}^{\frac{N}{n_p}-1}\sum_{j=2}^{l+1} M_j R$	-
MTTKRP sequence	ADATM	$2\sum_{i=1}^{n_p}\left(\sum_{l=2}^{N} M_l R + \sum_{l=1}^{\frac{N}{n_p}-1}\sum_{j=2}^{l+1} M_j\right) R$	$\sum_{i=1}^{n_p}\left(M^i_{CSF} + 8\sum_{l=\frac{N}{n_p}-n_i+1}^{\frac{N}{n_p}} M_l R\right)$
	SPLATT [156]	$2NMR$	M_{CSF}

Indices and values use "uint64_t" and "double" respectively. M_l is the number of fibers at the l^{th}-level of a CSF tree in figure 3.4(b), $M_{CSF} = 16\sum_{l=1}^{N} M_l$.

3.4.4 Strategy Guided Trade-off

Having predicted time and space, the framework searches the implementation under two strategy options: *performance-essential*, for the maximum performance; *space-efficient*, for the close-to-maximum performance but saving more space. In particular, for space-efficient strategy ADATM chooses the configuration with the smallest predicted space while having predicted performance no worse than some fraction of the optimal performance of all candidate configurations. (Our experiments use a fraction of 90%.)

3.4.5 Parallelization

Two parallel strategies are employed in ADATM: parallelizing among one mode and among multiple modes. Multiple-mode parallelism becomes possible because a partial MTTKRP can be parallelized among all the fibers of the saved intermediate tensors. For lower-order tensors with long modes, this strategy may not be attractive because parallelizing only one mode can expose enough parallelism for relatively small numbers of cores, as for current multicore architectures. Since multiple-mode parallelism is finer-grained than single-mode parallelism, for higher-order tensors with short modes, this strategy has performance advantages. Additionally, it would most likely map better to manycore co-processors (e.g., NVIDIA GPUs, Intel Xeon Phi processors), though we have not explored this possibility in this work.

3.5 Experiments and Analyses

Our experimental evaluation considers (a) reporting the speedup of AdaTM over the state-of-the-art SPLATT and Tensor Toolbox libraries; (b) analyzing the needed storage to obtain this speedup; (c) evaluating the space-time tradeoff to choose the optimal n_p; (d) assessing thread scalability and dimension scalability; and (e) verifying our model. Lastly, we apply our framework to CPD and show its performance.

3.5.1 Data Sets and Platforms

This evaluation uses the two platforms shown in table 3.3. We mainly show the results on the Xeon-based system because of its larger memory size and more cores, while the Core-based system is used to verify the portability of our model. All computations are performed in double-precision and the rank R is set to 16.

Table 3.3: Experimental Platforms Configuration

Parameters	Intel Xeon E7-4820	Intel Core i7-4770K
Microarchitecture	Westmere	Haswell
Frequency	2.0 GHz	3.5 GHz
#Physical cores	16	4
Memory size	512 GiB	32 GiB
Memory bandwidth	34.2 GB/s	25.6 GB/s
Compiler	gcc 4.4.7	gcc 4.7.3

We use two types of data. The first are third-order sparse tensors from real applications including Never Ending Language Learning (NELL) project [36] (nell1 and nell2 with *noun-verb-noun*) and data crawled from tagging systems [64] (deli with *user-item-tag*). (Refer FROSTT [151], the Formidable Repository of Open Sparse Tensors and Tools, for more details.) The second are higher-order, hyper-sparse tensors constructed from Electronic Health Records (EHR), with orders as high as 85. (In domains other than data analysis, constructing higher-order tensors to do low-rank approximation has been extensively used

to achieve good data compression [45, 70].) Additional details appear in table 3.4.

Table 3.4: Description of sparse tensors.

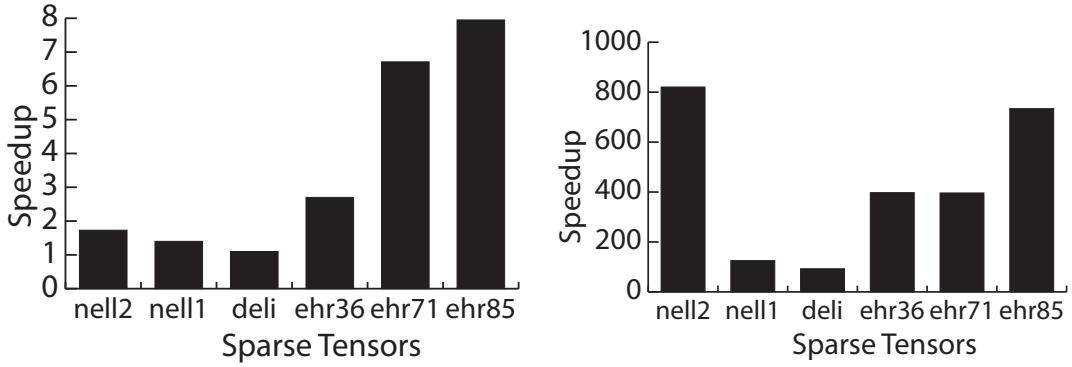
Tensors	Order	Max Mode Size	NNZ	Density
nell2	3	30K	77M	1.3×10^{-05}
nell1	3	25M	144M	3.1×10^{-13}
deli	3	17M	140M	6.1×10^{-12}
ehr36	36	19	11K	4.7×10^{-26}
ehr71	71	21	221K	1.4×10^{-55}
ehr85	85	21	920K	7.9×10^{-68}

3.5.2 Performance

We test the speedups of ADATM over SPLATT [157] and TENSOR TOOLBOX libraries [16] for an MTTKRP sequence in figure 3.9.[3] SPLATT v1.1.1 is tested under "SPLATT_CSF_TWOMODE" mode, which is usually the best case, without preprocessing nor tiling [4] , and Tensor Toolbox v2.6 is tested in MATLAB R2014a environment. ADATM achieves a speedup up to $1.7\times$ for the third-order tensors and up to $8.0\times$ for the higher-order tensors compared to the best parallel performance of SPLATT. The higher speedups on higher-order tensors in figure 3.9(a) show the advantage of ADATM with respect to the tensor order. Tensor deli merely show speedup since ADATM in mode-2 has worse threading scalability than SPLATT. Compared to TENSOR TOOLBOX, ADATM obtains $93 - 820\times$ speedups, because TENSOR TOOLBOX uses COO format and a less efficient and sequential MTTKRP algorithm. Tensor nell2 achieves an even higher speedup than higher-order tensors in figure 3.9(b) because of its best threading scalability among all tensors. (Its sequential speedup over Tensor Toolbox ($58\times$) is much less than that of higher-order tensors ($325 - 617\times$).) We mainly use SPLATT to evaluate our algorithm afterwards.

[3] We show speedups because the running time of the MTTKRP sequence varies a lot on these tensors.

[4] Cache tiling is beneficial cooperatively with tensor reordering in [156], which requires expensive preprocessing.



(a) SPLATT (b) Tensor Toolbox

Figure 3.9: Speedup of ADATM over SPLATT and Tensor Toolbox.

3.5.3 Analysis

Storage Table 3.5 shows the needed storage for all tensors, when achieving the best performance in figure 3.9. Traditional software (e.g., TENSOR TOOLBOX) uses COO format, while SPLATT proposed the more efficient CSF format. Since we use "CSF_TWOMODE" mode for good SPLATT performance, two CSF tensors are created (refer to [152] for details). Thus, some tensors in CSF format need more space than in COO format. ADATM's space is shown as "CSF+vCSF" by summing the space of CSF format for sparse tensors and vCSF format for semi-sparse tensors. We calculate the ratios of the ADATM's space to COO and CSF respectively in the right-most two columns. ADATM uses 102%-411% space of CSF and 78%-265% of COO, while achieving up to $8\times$ speedup in figure 3.9. We also calculated the space for tensor nell2 using the "avoid duplicated computation" method in [190], which stores the large dense matrices generated from Khatri-Rao product chain, 77952 MBytes extra space is needed. This is much larger than the storage size of ADATM (2581 MBytes). When ADATM is guided by "space-efficient" strategy, we can save some space without harming much performance.

Choosing n_p Theoretic optimal $n_p^* = 7$ is calculated from lemma 3.3.2 for tensor ehr85, figure 3.10 shows the relation between time and space for n_p in the range $[1, 7]$. $n_p = 0$ is the time and space of SPLATT. ADATM gets the shortest execution time on $n_p = 2$, when

Table 3.5: Storage size of sparse tensors.

	Storage Size (MBytes)			Ratios	
Dataset	COO	CSF	CSF+vCSF	/CSF	/COO
nell2	2290	2540	2581	102%	113%
nell1	4280	6430	8510	132%	199%
deli	4180	5570	11090	199%	265%
ehr36	3.04	1.94	7.97	411%	262%
ehr71	121	62	205	333%	169%
ehr85	604	200	470	236%	78%

$n_p > 2$ the running time of ADATM increases as well as its space. Because of this relation, ADATM can adapt the output configuration according to user-defined tradeoff strategy. Though the space on $n_p = 2$ is 236% of SPLATT's, ADATM achieves $6.4\times$ speedup.

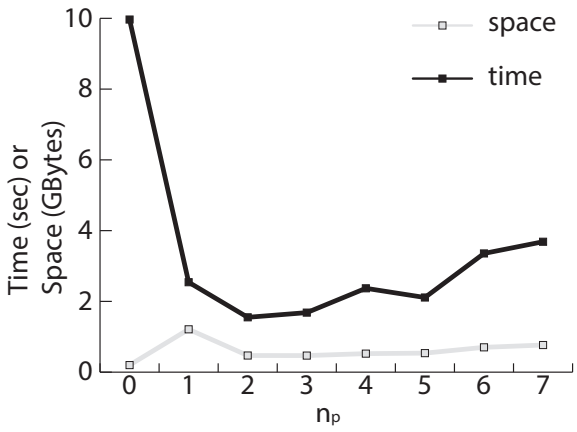
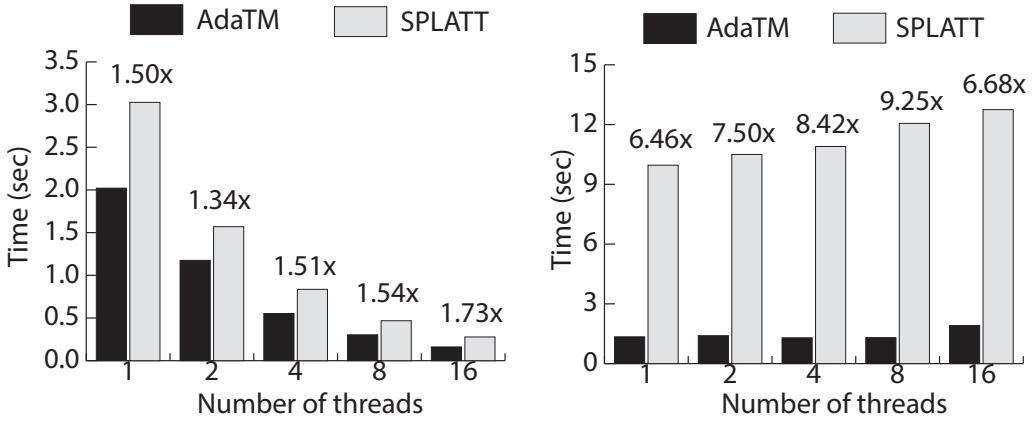


Figure 3.10: Time and space relation for ehr85.

Scalability We analyze thread scalability and dimension scalability. Figure 3.11 shows the thread scalability of ADATM and SPLATT on tensors nell2 and ehr85. The numbers above the bars are the speedup of ADATM over SPLATT when using the same number of threads. ADATM and SPLATT both obtain good scalability on nell2. However, SPLATT does not scale at all on ehr85, while ADATM gets the highest performance on 4 threads. The speedups in the two figures mainly increase with growing thread numbers, showing ADATM has better scalability using multiple-mode parallelism. Similar behavior is also observed from the other tensors, demonstrating that the thread scalability of higher-order

tensors needs further improvement.



(a) *nell2* (b) *ehr85*

Figure 3.11: Multithreading scalability of ADATM and SPLATT on nell2 and ehr85.

To better evaluate the dimension scalability, we use the method from [40] in TENSOR TOOLBOX to create synthetic higher-order sparse tensors. We generate eight sparse hypercubical tensors from 10 to 80th-order, all with 100,000 non-zeros and equal mode size 1000. The running time of ADATM and SPLATT is shown in figure 3.12. As tensor order grows, SPLATT's time increases much faster than ADATM's, which verifies ADATM's good speedup on higher-order tensors.

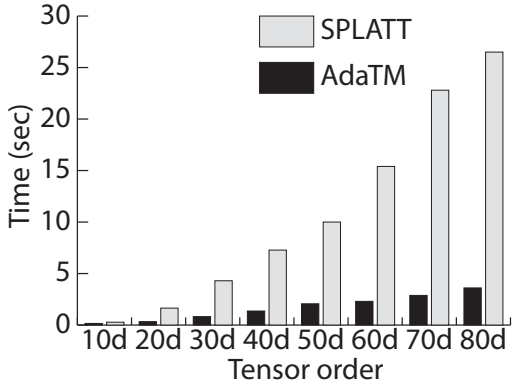


Figure 3.12: ADATM's dimension scalability on synthetic sparse tensors.

Model Analysis We test all possible n_ps of tensor ehr85 to record their actual time on the two platforms and compare with our predicted time in § 3.4. Figure 3.13 draws the

relative value by normalizing to each time value on $n_p = 1$. Model-predicted time can find the optimal n_p, which is 2, and predicts a similar trend to the actual time on the two platforms. Since the MTTKRP is dominated by TTM and q-TTM products which have relatively predictable behavior by timing a dense matrix, our model works well. However, since our model has not considered architecture characteristics, the prediction is constant for different platforms.

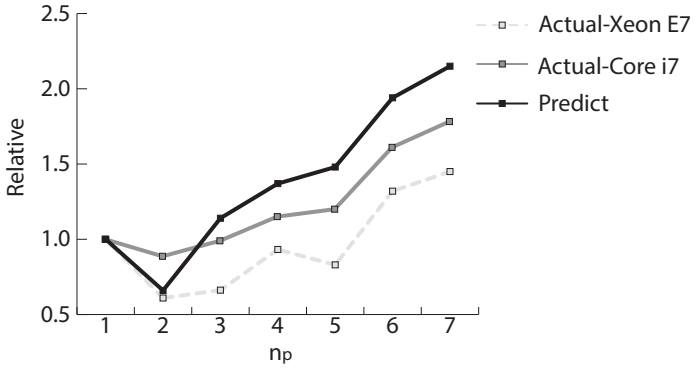


Figure 3.13: Accuracy of ADATM model on Xeon E7-4820 and Core i7-4770K.

3.5.4 CPD

We compare the running time of CP-ALS using ADATM and SPLATT in figure 3.14. Since the MTTKRP sequence dominates CP-ALS, ADATM shows good speedups especially on higher-order tensors. ADATM speedups CP-ALS by $4\% - 69\%$ for the third-order tensors and $2 - 8\times$ for the higher-order tensors. The speedups are similar to or slightly lower than those of MTTKRP sequence in figure 3.9(a). This figure shows ADATM is applicable to tensor decompositions in real applications.

3.6 Related Work

Researchers have successfully optimized a single MTTKRP operation through various methods. TENSOR TOOLBOX [16] and Tensorlab [171] implemented MTTKRP in the most popular COO format by multiple sparse tensor-vector products using MATLAB, with a high

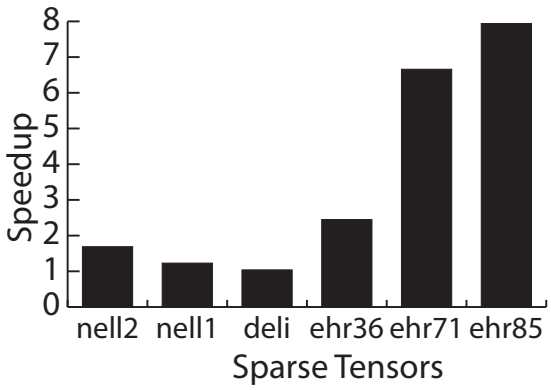


Figure 3.14: CP-ALS runtime using SPLATT and ADATM.

number of flops $\mathcal{O}(NMR)$. SPLATT [152, 156], the implementation achieving the highest performance so far, algorithmically improved MTTKRP by factoring out inner multiplications and proposed a more compressed CSF format, reducing MTTKRP to $\mathcal{O}(N^{\epsilon}MR)\,, \epsilon \in [0, 1)$ flops. GigaTensor [80] reformulated MTTKRP as a series of Hadamard products to utilize the massive parallelism of MapReduce. However, this algorithm is not suitable for multicore machines because of its high complexity ($\mathcal{O}(5MR)$ for third-order tensors). DFacTo [41] considered MTTKRP as a series of sparse matrix-vector multiplications for distributed systems. Though it has the same number of flops as SPLATT, the explicit matricization takes non-negligible time. HyperTensor [85] investigated fine- and coarse-grained distributed algorithms also for distributed systems, while its MTTKRP implementation is based on COO format. These approaches require $\mathcal{O}\left(N^{(1+\epsilon)}MR\right), \epsilon \in [0, 1]$ flops for the MTTKRP sequence. Our work is based on the SPLATT algorithm by memoizing intermediate results to explore data reuse and reduces this flop-complexity to $\mathcal{O}\left(\tilde{N}MR\right)$, where $\tilde{N}$ is generally much less than $N^{(1+\epsilon)}$.

Recently, S. Zhou et al. [190] and A. Phan et al. [137] exploited data reuse for the MTTKRP sequence by storing the results of dense Khatri-Rao products. The extra storage is $\Theta\left(I^{(N-1)}R\right)$ and $\Omega\left(I^{\frac{N}{2}}R\right)$ respectively for a hypercubical sparse tensor, which is extremely larger than the number of non-zeros and exponentially grows with tensor order N. ADATM efficiently stores intermediate semi-sparse tensors instead and in a very

compressed pattern to avoid redundant computation and also memory blowup. Baskaran et al. [21] and Kaya et al. [84] also brought similar ideas to save intermediate results in Tucker decomposition. However, our work does a detailed analysis on the most significant factor – the space and time tradeoff. Very recently, Kaya et al. [86] applied the above method in CP decomposition by using dimension trees and considered MTTKRP as a group of tensor-times-vector products. Our work uses TTM and q-TTM for better data locality, and ADATM allows user-defined strategy for the space-time tuning in our model-driven framework.

3.7 Summary

The methods underlying ADATM derive performance improvements from three key ideas. First, we consider not just a single MTTKRP, but the sequence as it arises in the context of CPD. Indeed, the lemmas in § 3.3 and the optimization method apply to other algorithms that might involve a Khatri-Rao product chain, such as the CP-APR algorithm [40]. Secondly, we develop an "any-space" memoization technique that permits a gradual tradeoff of storage for time. A user or higher-level library may therefore control our method according to whichever criterion is more important in a given application. Thirdly, we parameterize our algorithm and build a model-driven and user-guided framework for it. This technique gives end-user application developers some flexibility in how they employ our algorithm.

CHAPTER 4

A NOVEL SPARSE TENSOR FORMAT — HICOO

A central problem in sparse tensor computations is designing a data structure that is compact, locality-enhancing, and easy to integrate into applications. The problem of picking a sparse tensor data structure is similar to the classical one of how to choose a sparse matrix format. There are many options for sparse matrices that tradeoff size, speed, and "fit" to the non-zero structure of a given input matrix and requirements of the applications [12, 13, 30–33, 43, 88, 93, 105, 109, 110, 117, 120, 147, 172, 173, 178, 180–182, 187]. Similarly, there are several proposals for sparse tensor storage [15, 21, 103, 107, 152]. This chapter proposes a new alternative. We refer to it as *hierarchical coordinate* format, or HICOO (pronounced "haiku").[1]

The design of HICOO is motivated by two critical issues that affect one's choice of format: *compactness* and *mode orientation*. Compactness refers to keeping the total bytes small. Mode orientation is the idea that a format favors iteration of the tensor modes in a particular order, as we explain next.

First consider the simpler case of mode orientation for sparse matrices. We say that a matrix has two *modes*, or "axes", namely, its rows and its columns. Let us number the modes, referring to the rows as the first mode, or "mode 1," and the columns as "mode 2." If the matrix is sparse, you might store it in compressed sparse row (CSR) format [147], where each row is a sparse vector and rows are packed contiguously one after the other. You can randomly access any row i from only its index in $\mathcal{O}(1)$ time; however, to find an (i, j) element, you must search the sparse vector representing row i's non-zeros to find j. If one wishes simply to iterate over all non-zeros, as a sparse matrix-vector multiply might do, it is the most efficient to have an outer-loop over rows and an inner-loop over the

[1] At the time of this writing, this work has been accepted for publication [104].

sparse non-zeros within the row. Therefore, we say CSR is *oriented* first toward mode 1, then mode 2; its mode orientation may be denoted by $1 \prec 2$ (1 precedes 2), reflecting this loop-order structure. A compressed sparse column (CSC) matrix orients modes as $2 \prec 1$.

The idea of mode orientation generalizes for tensors. Consider an Nth-order tensor to be an N-way array, meaning it has N modes. CSR generalizes to N modes naturally; in the literature, this format is known as *compressed sparse fiber* (CSF) [152], which is parameterized by some mode orientation. For instance, a CSF-1 tensor has a mode orientation of $1 \prec 2 \prec 3 \prec \cdots \prec N$, with some ordering convention for other cases, CSF-k.

Mode orientation matters because sparse tensor analysis methods may require iterating over the modes in several orientations during the same computation [15]. The MTTKRP sequence in CPD is one motivating example. For a fixed storage format, iteration might be fast in one orientation but slow when the orientation switches. For a low-order (small-N) tensor, e.g., a matrix, it might be feasible to store multiple copies of the matrix to mitigate this effect. For example, one might store the matrix in both CSR and CSC formats and use the appropriate one when it applies, so the order no longer matters. But for tensors, as the order N grows, a multiple-copy strategy can become infeasible.

One can avoid mode orientation altogether. The simplest, and arguably most popular, storage format for sparse tensors is COOrdinate (COO) format [15, 77, 171]. COO records each non-zero as a tuple, $(i_1, i_2, \ldots, i_N; v)$, where i_k is an index coordinate and v is the non-zero value. Thus, it has a neutral or "agnostic" mode orientation. However, the price is that it is less compact than formats like CSF, which can exploit mode orientation to reduce the average amount of index metadata (i.e., the i_k values) per non-zero.

Our proposed HICOO format tries, heuristically, to achieve *both* compactness and neutral mode orientation. By contrast, essentially all proposed formats [15, 107, 152] can achieve compactness but not, simultaneously, neutral mode orientation, at least not without paying a performance penalty. This situation includes CSF [152] from above, as well as the more recent flagged-coordinate (F-COO) format [107]. For example, consider the com-

monly occuring tensor operation MTTKRP (§ 2.2.4). Performing it in a given mode using the CSF representation oriented toward a different mode can suffer $3\times$ slowdown for tensor choa in mode 2. A tensor operation for an F-COO representation can only be performed in a designated mode orientation implied in its representation (§ 4.1).

Our claimed contributions of HICOO and the present study may be summarized as follows.

- We first compare and analyze COO, CSF, and F-COO formats, along the criteria of compactness and mode orientation, as well as their expected behavior on real tensor computations, such as the matricized tensor-times-Khatri-Rao product (MTTKRP), which motivates this work (§ 4.1).

- We describe HICOO, which compresses tensor indices in units of sparse tensor blocks and exploits shorter integer types to express offsets within the blocks. Since HICOO has a neutral mode orientation, only one HICOO representation is needed (§ 4.2).

- We accelerate MTTKRP on multicore CPU architectures based on HICOO. Using a superblock scheduler and two parallelization strategies, our parallel HICOO-MTTKRP exhibits better thread scalability than COO- and CSF-based MTTKRPs (§ 4.3).

- Overall, HICOO achieves up to $23.0\times$ ($6.8\times$ on average) speedup over COO and $15.6\times$ ($3.1\times$ on average) speedup over CSF for a single MTTKRP operation; it can also use up to $2.5\times$ less storage than COO format and comparable storage with only one CSF representation. When MTTKRP is integrated into a complete tensor decomposition algorithm (known as "CPD"), the HICOO-based implementation is also faster than COO- and CSF-based implementations (§ 4.4).

- HICOO and these CPD and MTTKRP algorithms are implemented in PARTI! library (will be discussed in Chapter 8).

Table 4.1 summarizes the symbols and notations only in Chapter 4, other general symbols are listed in table 2.1.

Table 4.1: List of symbols and notations in Chapter 4.

Symbols	Description
Input parameters	
M_l	#Nodes at level l of a CSF tree of tensor $\mathcal{X}$
P	#CPU threads
S_{cache}	Cache size
HICOO Data structures	
einds	Element indices of HICOO, β_{byte} bits
binds	Block indices of HICOO, β_{int} bits
bptr	Block pointers of HICOO, β_{long} bits
lptr	Superblock pointers of HICOO, β_{long} bits
lschr	Superblock scheduler of HICOO, β_{int} bits
HICOO parameters	
L	Tensor superblock size
B	Tensor block size, $B \ll L$
$\overline{c}$	Average slice size, $\overline{c} = \frac{M}{I_n}$
n_l	#Nonzero tensor superblocks
n_b	#Nonzero tensor blocks
α_b	Block ratio, $\alpha_b = \frac{n_b}{M}$
$\overline{M_b}$	Geometric mean of #Nonzeros per tensor block
$\overline{c_b}$	Average slice size per tensor block, $\overline{c_b} = \frac{\overline{M_b}}{B}$

4.1 Sparse Tensor Format Comparison

Let us regard COO as the baseline storage format and compare it analytically to two state-of-the-art formats: CSF [152] and F-COO [107], described in § 2.1.3. We consider general unstructured sparse tensors and assess the formats in terms of their storage and behavior for the MTTKRP operation, e.g., floating-point operations or flops, memory traffic, and arithmetic intensity. The results motivate the present work on HICOO.

For simplicity, this analysis assumes an Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ with M non-zeros. We assume an integer index needs β_{int} bits and a non-zero floating-point value takes β_{float} bits. Lastly, we use β_{long} bits for a pointer to index all non-zeros in a very large tensor. The summarized comparison of Table 4.2 substitutes values for them that reflect typical choices based on standard primitive types.

Table 4.2: The analysis of tensor formats and their MTTKRP algorithms for a third-order tensor ($N = 3$) with M nonzero entries. The word size parameters are $\beta_{\text{int}} = 32$, $\beta_{\text{long}} = 64$, $\beta_{\text{byte}} = 8$, and $\beta_{\text{float}} = 32$ bits for single-precision floating-point values and discarding insignificant items.

	Data Structure		MTTKRP Behavior		
Format	Index Space (Bits)	Update Needed?	Work (Flops)	Memory Access (Bytes)	Arithmetic Intensity (AI)
COO	$96M$	NO	$3MR$	$12MR$	$1/4$
F-COO	$65M$	YES	$3MR$	$12MR$	$1/4$
CSF	$[32M, 128M]$	YES[1]	$[2MR, 4MR]$	$[8MR, 16MR]$	$1/4$
HICOO	$[24M, 184M]$	NO	$3MR$	$min\{\frac{12}{c_b}, 12\}MR$	$max\{\frac{1}{4}, \frac{c_b}{4}\}$

[1] MTTKRPs in all tensor modes can use less CSF representations than tensor order or even only one CSF representation with performance payoff.

4.1.1 Summary

We begin with an overall summary of our observations, followed by the detailed analysis.

- **Observation 1:** *CSF generally achieves the best compression for a single representation, but it is worse than COO by storing multiple representations for all modes.* However, using only one CSF representation could suffer a performance penalty. (Table 4.2, column "Index Space")

- **Observation 2:** *CSF and F-COO need extra time and space to construct another representation for the same tensor operation but in a different mode.* That is, neither format is, on its own, neutral to mode orientation. (See table 4.2, "Update Needed?")

- **Observation 3:** MTTKRP *implementations based on COO, CSF, and F-COO formats are expected to have arithmetic intensities of approximately 0.25 flops per byte, which means they will be memory-bound on most platforms.* Reducing memory traffic by enhancing data locality could be beneficial for MTTKRP. (See table 4.2, "Arithmetic Intensity")

4.1.2 COO Format

The COOrdinate (COO) format is the simplest yet arguably most popular format, which is described in § 2.1.3 and figure 2.5(a). It stores each non-zero value along with all of its position indices. COO does not favor any mode over the others, which gives it a neutral mode orientation.

COO format analysis Storing all the indices uses

$$S_{\text{coo}} = N \cdot M \cdot \beta_{\text{int}} \tag{4.1}$$

bits.

Algorithm 6 COO-MTTKRP algorithm([15]).

Input: A third-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, dense matrices $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$;
Output: Updated dense matrix $\tilde{\mathbf{A}} \in \mathbb{R}^{I \times R}$;

▷ $\tilde{\mathbf{A}} \leftarrow \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$

1: **for** $x = 1, \ldots, M$ **do**
2: $\quad i = \text{inds}(x, 1), j = \text{inds}(x, 2), k = \text{inds}(x, 3)$;
3: $\quad$ **for** $r = 1, \ldots, R$ **do**
4: $\quad\quad \tilde{A}(i, r) += \text{val}(x) C(k, r) B(j, r)$
5: **return** $\tilde{\mathbf{A}}$;

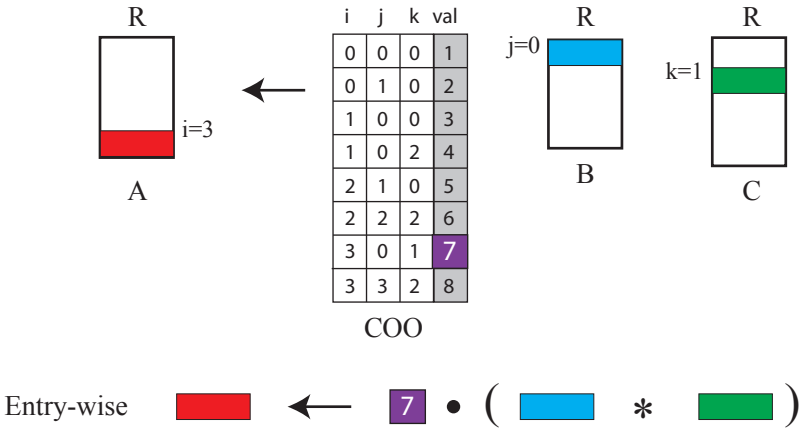


Figure 4.1: A COO-MTTKRP example in mode 1 showing the operations on one non-zero tensor entry 7. Its corresponding matrix rows are plotted as solid boxes inside.

COO-MTTKRP For illustration purposes, the pseudocode of COO-MTTKRP for third-order tensors appears in Algorithm 6 and its graphical plot is in figure 4.1. MTTKRP multiplies every non-zero entry at position (i, j, k) with row-j of $\mathbf{B}$ and row-k of $\mathbf{C}$, then reduces the rows to row-i of $\tilde{\mathbf{A}}$. The three rows may be irregularly distributed in the matrices depending on the sparsity pattern of $\mathcal{X}$.

For a general N, a rank-R COO-MTTKRP performs

$$\text{Flops}_{\text{coo}} = N \cdot M \cdot R \tag{4.2}$$

flops, where N operations per non-zero for a matrix column consist of $(N-1)$ multiplications and one addition. Assume no reuse exists in $\tilde{\mathbf{A}}$, $\mathbf{B}$, and $\mathbf{C}$,[2] its memory traffic is

$$\text{Bytes}_{\text{coo}} = \frac{M}{8}(\underbrace{N\beta_{\text{int}}}_{\text{indices}} + \underbrace{\beta_{\text{float}}}_{\text{values}} + \underbrace{NR\beta_{\text{float}}}_{\text{matrices}}) \tag{4.3}$$

bytes, by counting the instant read and write operations of $\tilde{\mathbf{A}}$ as a one-time memory access. Therefore, its arithmetic intensity (AI), or ratio of the number of flops to the number of memory accesses, is about $1/4$ when $\beta_{\text{int}}, \beta_{\text{float}} = 32$ bits (as listed in Table 4.2).

4.1.3 CSF Format

CSF (compressed sparse fiber) is a hierarchical, fiber-centric format (described in § 2.1.3 and figure 2.5(b)) implemented in the SPLATT package [157], which has a strong mode orientation.

For an Nth-order tensor, N CSF trees would be needed for the best performance if the tensor operation of interest needs to be iterated over the tensor multiple times, once in each mode, as stated in [154]. An analogy would be a computation that needed both parallel matrix-vector and matrix-transpose-vector multiplies without write conflicts; a simple and

[2] Even though this is the worst case of memory traffic, it is reasonable as an approximation of real sparse tensors which typically have a very small degree of data reuse in MTTKRP. Similar assumption is also made for the following CSF and F-COO analysis.

fast approach would be to store the matrix in both CSR and CSC formats at the cost of doubling the space as suggested in [174]. However, storing multiple CSF trees consumes a large extra storage space (as will be shown in Table 4.4).

CSF format analysis The CSF format uses

$$S_{\text{csf}} = \underbrace{\sum_{l=1}^{N-1}(M_l+1)\cdot\beta_{\text{long}}}_{\text{pointers}} + \underbrace{\sum_{l=1}^{N} M_l\cdot\beta_{\text{int}}}_{\text{indices}} \tag{4.4}$$

bits for a *single* CSF tree, where M_l represents the number of nodes at level l in Figure 2.5(b). $M_l = M$ when $l = N$, and $M_l = I_n$ when $l = 1$. Besides indices for all N levels, CSF stores pointers at non-leaf levels to index the beginning locations of every sub-tree, i.e., sub-tensor. These pointers are analogous to the row pointers in CSR for sparse matrices. If a tensor has $M_l \ll M, l = 1, \ldots, N-1$, CSF achieves good compression; otherwise, if $M_l \approx M$, it may need even more storage than COO because of the overhead of storing additional pointers with more bits (β_{long} instead of β_{int}). These two extreme cases are shown in Table 4.2 as lower and upper bounds. When maintaining multiple CSF trees for higher MTTKRP performance, the total storage will be their sum.

CSF-MTTKRP CSF-MTTKRP performs

$$\text{Flops}_{\text{csf}} = 2R\sum_{l=2}^{N} M_l \tag{4.5}$$

flops [156]. It incurs

$$\text{Bytes}_{\text{csf}} = \frac{1}{8}[\underbrace{\beta_{\text{long}}\sum_{l=1}^{N-1} M_l}_{\text{pointers}} + \underbrace{\beta_{\text{int}}\sum_{l=1}^{N} M_l}_{\text{indices}} + \underbrace{M\beta_{\text{float}}}_{\text{values}} + \underbrace{(2R\sum_{l=2}^{N} M_l)\beta_{\text{float}}}_{\text{matrices}}] \tag{4.6}$$

bytes of memory accesses [3] [152, 156]. Therefore, its arithmetic intensity (AI) is also about $1/4$ when $\beta_{\text{int}}, \beta_{\text{float}} = 32$ bits, $\beta_{\text{long}} = 64$ bits.

4.1.4 F-COO Format

Flagged-COOrdinate (F-COO) format, recently proposed in [107], which is described in § 2.1.3 and figure 2.5(c) as an example of a F-COO representation for mode-1 MTTKRP. Intuitively, bf and sf indicate any changes in the index mode(s), then FCOO-MTTKRP uses segmented scan primitive to avoid locks or atomic operations.

F-COO format analysis We only analyze F-COO for the MTTKRP operation in this work, interested readers could refer to [107] for other scenarios, e.g., tensor-times-matrix multiplication (TTM). The F-COO format uses

$$S_{\text{fcoo}} = M[\underbrace{(N-1)\beta_{\text{int}}}_{\text{indices}} + \underbrace{1}_{\text{bit-flag}} + \underbrace{\frac{1}{M_{thread}}}_{\text{start-flag}}] \tag{4.7}$$

bits to hold the indices, where M_{thread} is the number of non-zeros assigned to a thread. It is strongly mode-oriented, thus a F-COO representation is required for every mode of a tensor operation.

F-COO MTTKRP MTTKRP based on F-COO performs

$$\text{Flops}_{\text{fcoo}} = N \cdot M \cdot R \tag{4.8}$$

[3] CSF-MTTKRP in SPLATT uses a R-array to accumulate intermediate results of inter-nodes. Since this small R-array could be easily cached between two adjacent node levels, only one of the two reads is counted.

flops, which is the same as COO-MTTKRP. It moves

$$\text{Bytes}_{\text{fcoo}} = \frac{M}{8}\left[\underbrace{(N-1)\cdot\beta_{\text{int}}+1+\frac{1}{M_{thread}}}_{\text{indices and flags}} + \underbrace{N\cdot R\cdot\beta_{\text{float}}}_{\text{matrices}}\right] \tag{4.9}$$

bytes of data to and from memory. Therefore, its AI is about $1/4$ when $\beta_{\text{int}}, \beta_{\text{float}} = 32$ bits.

From the above three observations and detailed analysis, we propose a new sparse tensor format HICOO to overcome the drawbacks of current formats, maintain neutral mode orientation, meanwhile, pursue higher performance by optimizing memory locality.

4.2 HICOO Format

HICOO stores a sparse tensor in a sparse-blocked pattern with a pre-specified block size B, meaning in $B \times \cdots \times B$ blocks (only hypercubical blocks are considered in this work). It represents every block by compactly storing its non-zero triples using fewer bits. A tensor is sorted and then partitioned and compressed by every mode into sized-B chunks, resulting in at most $\frac{I_1}{B} \times \cdots \times \frac{I_N}{B}$ (assume all I_ns are dividable by B) non-zero tensor blocks. Figure 4.2 shows the same third-order tensor example as figure 2.5 given $2 \times 2 \times 2$ blocks ($B = 2$). For a third-order tensor, bi, bj, bk are *block indices* in β_{int} bits, indexing tensor blocks, and ei, ej, ek are *element indices* in β_{byte} bits, indexing non-zeros within a tensor block. A bptr array in β_{long} bits stores the pointers of every block's beginning locations, and val saves all the non-zero values, which is the same as COO's val array. HICOO treats every mode equally and does not assume any mode order, these preserve the neutral mode orientation of COO.

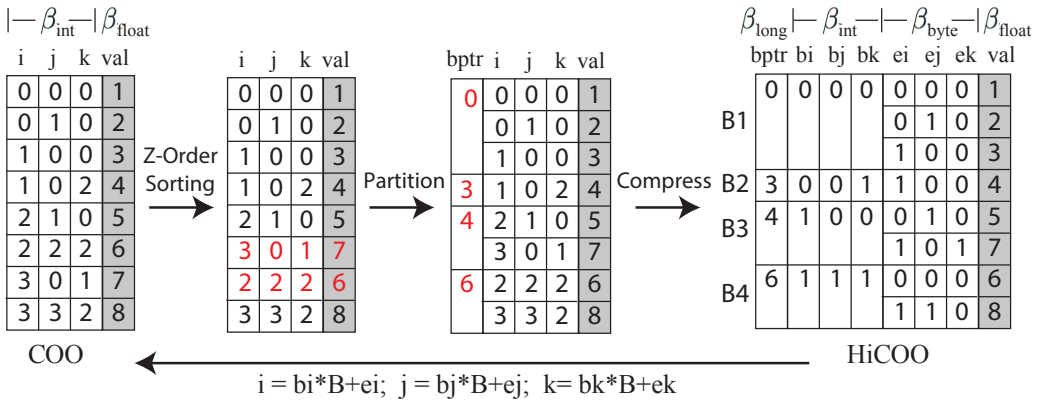


Figure 4.2: The conversion between COO and HICOO formats for an example third-order tensor. HICOO uses $2 \times 2 \times 2$ blocks ($B = 2$) with word sizes marked above.

4.2.1 Conversion

Sorting, partitioning, and compression are the three steps to convert from a COO tensor to a HICOO tensor. We first sort all non-zeros of a COO tensor in Z-Morton order [119] using a variation of quick sort. A Morton key is computed from non-zero indices and is used for the comparison of sorting. In Figure 4.2, the sorted COO tensor switches two non-zero entries (marked in red). We then partition the sorted tensor into sparse tensor blocks according to the given block size B and record the block pointers bptr simultaneously. Since we limit the block size to a power-of-two constant, this partitioning maintains the sorted Z-order between tensor blocks and among the non-zeros within a block. Lastly, we compress COO indices into block and element indices correspondingly. By having a HICOO tensor, no need to explicitly convert it back to a COO tensor. The COO indices of a non-zero entry can be calculated from $i = \mathrm{bi} \cdot B + \mathrm{ei}$, $j = \mathrm{bj} \cdot B + \mathrm{ej}$, and $k = \mathrm{bk} \cdot B + \mathrm{ek}$.

In the HICOO format, Z-Morton sorting contributes better data locality for tensor algorithms, while compressed indices save the storage space of a sparse tensor and also reduce the memory bandwidth of tensor access.

4.2.2 Improvement of CSB

Our proposed Hierarchical COOrdinate (HICOO) format may be viewed as an extension of the Compressed Sparse Blocks (CSB) format for sparse matrices [30]. Two critical features of CSB inspired us: 1) it targets large matrices with hypersparse *matrix blocks*; 2) CSB is neutral to mode orientation and allows efficient computation of both Sparse Matrix-Vector Multiplication (SpMV) and Sparse Matrix-Transpose-Vector Multiplication (SpMTV) using a single CSB representation.

One distinction between HICOO and CSB is that the latter uses relatively larger *matrix blocks*. In CSB, block sizes are typically around $\sqrt{I} \times \sqrt{I}$ for an $I \times I$ sparse matrix [30]. By contrast, we find that smaller blocks are more suitable for sparse tensors, both for reasons related to better cache usage and better support for higher-order tensor operations. However, small blocks pose two issues of a straightforward extension of CSB: 1) First, CSB is not storage-efficient for small blocks as stated in [30]. Small blocks certainly lead to more compressed non-zero indices by being capable of using fewer bits, however, the storage of block indices, which is saved contiguously as a dense array, increases much faster. Therefore, the overall storage of CSB is not beneficial from small blocks. 2) Secondly, small blocks imply a relatively fine-grained parallelism. On our target multicore platforms, heavyweight CPU threads are not efficient to schedule a huge number of threads with a small workload of each.

To solve these challenges, HICOO improves from CSB idea in two aspects.

- First, HICOO further compacts block indices, thus requiring even less storage space than CSB. We compact block indices in coordinate pattern to control their storage rise for small blocks and also uses fewer bits when possible.

- Second, for efficient CPU multithreading, HICOO uses a two-level blocking strategy and a small amount of extra space to save runtime scheduling information. We group a set of small blocks into a large yet logical *superblock*. The blocks within a

superblock are always scheduled together and assigned to a single thread. Within a superblock, we physically store non-zeros in the same pattern as shown in figure 4.2. This two-level blocking strategy will be better explained in § 4.3.2 since it is more related to algorithm parallelization.

4.2.3 Analysis

Our analysis of HICOO will be expressed in terms of parameters listed in table 4.1. We first explain these parameters and give the format analysis afterwards.

The *Average Slice Size ($\overline{c}$)* is a tensor-dependent parameter. It is an analogy of "row length" of a sparse matrix. $\overline{c}$ is the average slice size in a particular mode n, $\overline{c} = \frac{M}{I_n}$. $\overline{c}$ could vary considerably, from being a constant ($\overline{c} = \mathcal{O}(1)$) if there are only a few non-zeros per slice, to being as large as $\overline{c} = \mathcal{O}(I^2)$ if slices are dense. The value $\overline{c}$ effectively measures non-zero density, especially for irregularly shaped sparse tensors.

The *Number of Tensor Blocks (n_b)* depends on the input tensor and HICOO-specific block size B. The example in figure 4.2 has $n_b = 4$ tensor blocks.

The *Block Ratio (α_b)* is the ratio of the number of tensor blocks to the number of total non-zero elements, $\alpha_b = \frac{n_b}{M}$. Block ratio directly affects the storage size of HICOO, which will be shown in Equation (4.10). For a given sparse tensor with a fixed M, α_b is not solely determined by the block size B, but also related to non-zero distribution.

The *Geometric Mean of Numbers of Non-zeros per Block ($\overline{M_b}$)* depends on non-zero distribution and block size B. We choose geometric mean because it is sensitive to skewed data and aware of unevenness distribution. From our experiments on real tensors in table 4.3, the numbers of non-zeros per block are generally skewed downward, with many small values and a few large ones, and unevenly distributed. The four tensor blocks in figure 4.2 consist of $3, 1, 2, 2$ non-zeros respectively, thus $\overline{M_b} = 1.9$.

The *Average Slice Size per Tensor Block ($\overline{c_b}$)* is analogous to $\overline{c}$: it is the average slice size in a block, $\overline{c_b} = \frac{\overline{M_b}}{B}$. The value $\overline{c_b}$ reflects the non-zero density of a block. It is also

crucial to MTTKRP performance (details in § 4.3). Note that $\overline{c} = \mathcal{O}(1)$ (or $\mathcal{O}(I^2)$) does not necessarily mean $\overline{c_b} = \mathcal{O}(1)$ (or $\mathcal{O}(B^2)$) because of potential nonuniform local and global non-zero distributions.

The HICOO format uses

$$S_{\text{hicoo}} = \underbrace{(n_b + 1) \cdot \beta_{\text{long}}}_{\text{bptr}} + \underbrace{N \cdot n_b \cdot \beta_{\text{int}}}_{\text{block indices}} + \underbrace{N \cdot M \cdot \beta_{\text{byte}}}_{\text{element indices}} \approx M[\alpha_b \cdot \beta_{\text{long}} + \alpha_b N \cdot \beta_{\text{int}} + N \cdot \beta_{\text{byte}}] \tag{4.10}$$

bits of storage. The index space shown in table 4.2 takes two extreme values 0 and 1 of α_b to get its lower and upper bounds. This amount will be smaller than COO when $S_{\text{hicoo}} < S_{\text{coo}}$, or

$$\alpha_b < \frac{\beta_{\text{int}} - \beta_{\text{byte}}}{\beta_{\text{int}} + \frac{1}{N}\beta_{\text{long}}}. \tag{4.11}$$

For a third-order tensor ($N = 3$) with $\beta_{\text{int}} = 32$ bits, $\beta_{\text{long}} = 64$ bits, $\beta_{\text{byte}} = 8$ bits, then $\alpha_b < 0.45$. This means if there are more than 2 non-zeros per block on average, theoretically, HICOO consumes less space than COO. The threshold of α_b grows with tensor order N, i.e., the sparsity restriction of HICOO becomes looser with increasing dimensionality.

Overall, HICOO takes small storage space, explicitly exposes a better data locality for every tensor mode, and has a neutral mode orientation. HICOO, as a general sparse tensor format, is able to support diverse types of tensor operations and various computing platforms. In this chapter, we choose one of the most important tensor operations MTTKRP as a proof-of-concept.

4.3 HICOO-MTTKRP Algorithms on Multicore CPUs

We use the HICOO format for the MTTKRP operation and introduce our optimization methods for sequential and multicore parallel algorithms.

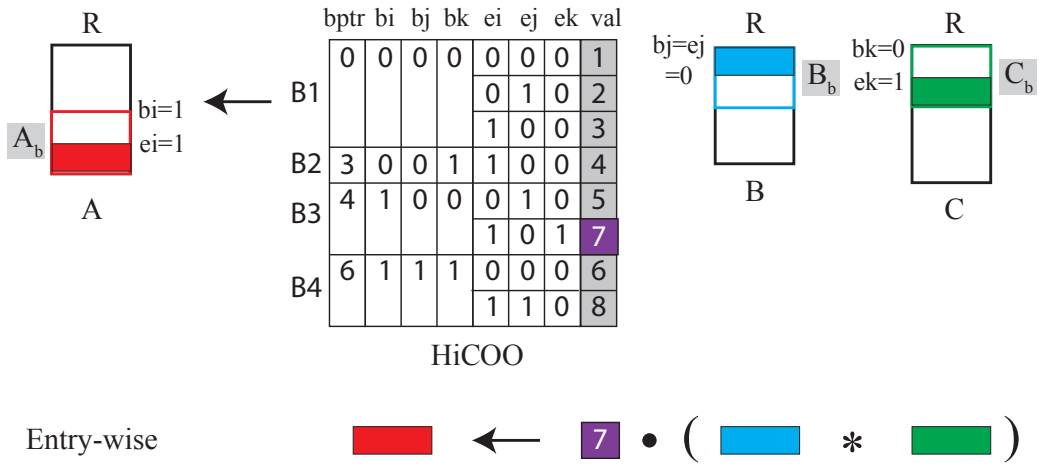


Figure 4.3: A HICOO-MTTKRP example in mode 1 showing the operations on one non-zero tensor entry 7. Its corresponding matrix blocks $\mathbf{A}_b, \mathbf{B}_b, \mathbf{C}_b$ are shown as bounded blank boxes, and its corresponding matrix rows are plotted as solid boxes inside.

4.3.1 Sequential Algorithm

Figure 4.3 depicts HICOO-MTTKRP algorithm on the example $4 \times 4 \times 3$ HICOO tensor from figure 4.2. Taking the non-zero entry with a value 7 in block $B3$ as an example, its block indices are $(1, 0, 0)$ and element indices are $(1, 0, 1)$. Block indices $(\mathrm{bi}, \mathrm{bj}, \mathrm{bk})$ identify the beginning locations of blocked matrices $\mathbf{A}_b, \mathbf{B}_b, \mathbf{C}_b$ (marked as blank boxes) by offsetting $\mathrm{bi} \cdot B, \mathrm{bj} \cdot B, \mathrm{bk} \cdot B$ rows from $\mathbf{A}, \mathbf{B}, \mathbf{C}$. Every tensor block only accesses blocked matrices $\mathbf{A}_b, \mathbf{B}_b, \mathbf{C}_b \in \mathbb{R}^{B \times R}$. Then the non-zeros within a block can be indexed only by element indices $(\mathrm{ei}, \mathrm{ej}, \mathrm{ek})$. The matrix rows needed by this non-zero element are plotted as solid boxes. At the bottom of figure 4.3, the two rows of $\mathbf{C}$ and $\mathbf{B}$ first do element-wise product, then their resulting vector is scaled by the non-zero value 7 to update one row of $\mathbf{A}$. We use SIMD directives to accelerate the vector operations of each non-zero entry. Sequential HICOO-MTTKRP algorithm for a third-order tensor is shown in Algorithm 7.

If block size B is configured to keep $\mathbf{A}_b, \mathbf{B}_b, \mathbf{C}_b$ cached in local memory, they can be re-used multiple times without memory transfers. Assume the number of non-zeros of a tensor block is $\overline{M_b}$, and every slice in the block has an equal size $\overline{c_b}$, thus the memory traffic

Algorithm 7 Sequential HICOO-MTTKRP algorithm.

Input: A third-order HICOO sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, dense matrices $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$, block size B;
Output: Updated dense matrix $\tilde{\mathbf{A}} \in \mathbb{R}^{I \times R}$;

▷ $\tilde{\mathbf{A}} \leftarrow \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$

1: **for** $b = 1, \ldots, n_b$ **do** ▷ block b
2: $bi = \text{binds}(b, 1)$, $bj = \text{binds}(b, 2)$, $bk = \text{binds}(b, 3)$;
3: $\mathbf{A}_b = \mathbf{A} + bi \cdot B \cdot R$; $\mathbf{B}_b = \mathbf{B} + bj \cdot B \cdot R$; $\mathbf{C}_b = \mathbf{C} + bk \cdot B \cdot R$;
4: **for** $x = bptr[b], \ldots, bptr[b+1] - 1$ **do** ▷ entry x
5: $ei = \text{einds}(x, 1)$, $ej = \text{einds}(x, 2)$, $ek = \text{einds}(x, 3)$
6: **for** $r = 1, \ldots, R$ **do**
7: $\tilde{A}_b(ei, r) += \text{val}(x) C_b(ek, r) B_b(ej, r)$
8: **return** $\tilde{\mathbf{A}}$;

of one blocked matrix is $NR \min\{B, \overline{M_b}\}\beta_{\text{float}}$ bits. When $\overline{M_b} > B$, or equally $\overline{c_b} > 1$, at most NBR values are transferred from memory for this blocked matrix because all its values are already cached after the first transfers; otherwise, if $\overline{c_b} < 1$, all $NR\overline{M_b}$ values will be transfered from memory as COO-MTTKRP and no reuse for them. Tensor blocking exploits data locality in memory transfers for the matrix operands, and also benefits the parallel algorithms that follow.

Analysis. Sequential HICOO-MTTKRP algorithm has

$$\text{Flops}_{\text{hicoo}} = N \cdot M \cdot R \tag{4.12}$$

flops, identical to COO-MTTKRP algorithm in Equation (4.2). Since HICOO-MTTKRP exploits matrix reuse, its memory traffic is

$$\begin{aligned}
\text{Bytes}_{\text{hicoo}} =& \frac{n_b}{8}[\underbrace{\beta_{\text{long}}}_{\text{one bptr value}} + \underbrace{2N \cdot \beta_{\text{int}}}_{\text{block indices and blocked matrices pointers}} \\
&+ \underbrace{\overline{M_b}(N \cdot \beta_{\text{byte}} + \beta_{\text{float}})}_{\text{element indices and values per block}} + \underbrace{NRmin\{B, \overline{M_b}\} \cdot \beta_{\text{float}}}_{\text{blocked matrix values}}] \\
&\approx \frac{M}{8}[\alpha_b \cdot \beta_{\text{long}} + 2\alpha_b N \cdot \beta_{\text{int}} + N \cdot \beta_{\text{byte}} + \beta_{\text{float}} \\
&+ NR \min\{\frac{1}{\overline{c_b}}, 1\} \cdot \beta_{\text{float}}] \\
&\approx \frac{M}{8}[2\alpha_b \cdot \beta_{\text{int}} + \beta_{\text{byte}} + R \min\{\frac{1}{\overline{c_b}}, 1\} \cdot \beta_{\text{float}}]N
\end{aligned} \tag{4.13}$$

bytes, where $n_b \times \overline{M_b} \approx M,\ \alpha_b = \frac{n_b}{M}$. Generally, the last term of loading factor matrices

dominates its memory traffic. Therefore, its arithmetic intensity (AI) is about $max\{\frac{1}{4}, \frac{\overline{c_b}}{4}\}$ when $\beta_{\text{int}}, \beta_{\text{float}} = 32$ bits, $\beta_{\text{byte}} = 8$ bits, which is listed in table 4.2. When $\overline{c_b} > 1$, HICOO-MTTKRP has higher arithmetic intensity than all the state-of-the-art MTTKRP algorithms.

4.3.2 Parallel Algorithm

As mentioned in § 4.2, our small tensor blocks are good for data locality but have parallel issues on multicore CPUs. Parallelizing small tensor blocks of sequential HICOO-MTTKRP (Line 1 in Algorithm 7) occurs a huge number of small workloads per thread, which is not efficient for heavyweight CPU threads. Besides, parallelizing either the first (line 1) or the second loop (line 4) leads to write conflicts, multiple threads may write the same blocked matrix $\tilde{\mathbf{A}}_b$ or even the same row of it. Because the rank R is very small, parallelizing the R-loop (line 6) is not efficient and causes false-sharing for $\tilde{\mathbf{A}}_b$. A simple solution is to use expensive locks or atomic operations to protect $\tilde{\mathbf{A}}_b$. Our work completely removes write conflicts by taking advantage of pre-knowledge from HICOO construction process.

Logical Superblocks. We increase the workload granularity of scheduling and employ two-level blocking that groups small blocks into larger ones which we call *superblocks*. A superblock is essentially a "logical" subtensor that can potentially consist of many small blocks. During HICOO format conversion, we first extract $L \times \cdots \times L$ non-zero superblocks then treat each superblock as an independent tensor to convert it to the physical HICOO format with $B \times \cdots \times B$ blocks ($L \geq B$). An additional array lptr is included to store the beginning pointers of non-zero superblocks, with its size n_l, the number of superblocks.

Superblock Scheduling. Since HICOO systematically partitions a Z-order sorted tensor into superblocks, data race information can be recorded and then used to guide parallel MTTKRP execution. A superblock scheduling table lschr is constructed to indicate write-conflict information between them. Generally, the overhead of storing lptr and lschr are negligible compared to the other components of HICOO, e.g., einds, binds, because of the small number of superblocks. Different superblock sizes will generate different scheduling

tables.

Figure 4.4 shows a lschr table for mode-1 MTTKRP. In the y-direction, superblocks (e.g., 0, 1, 2, 3) have no write conflicts because they are updating different regions of $\tilde{\mathbf{A}}$. These superblocks can be independently parallelized, therefore, a reasonable large number of superblocks in the y-direction is preferable. In the x-direction, superblocks (e.g., 0, 4, 8) have potential write conflicts (shown as two-way arrows) that are, therefore, avoidable if they are executed sequentially. In this figure, three iterations are needed to serialize all the 10 superblocks. The larger the number of iterations, the stronger dependence shown in this MTTKRP. With this superblock scheduler, we iterate superblocks in two loops: one for independent superblocks, the other for iterations; and choose a strategy to parallelize one of them.

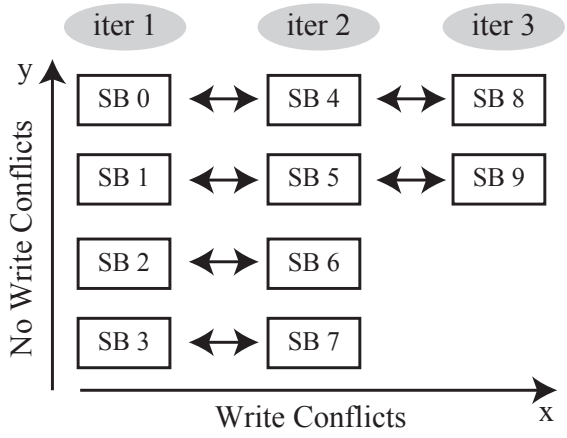


Figure 4.4: Superblock scheduling table for mode-1 MTTKRP. Write conflicts are shown as two-way arrows.

Parallel Strategies. The easy case is that we have reasonable large independent superblocks to directly parallelize. However, the number of independent superblocks depends on the product mode size (i.e., I_n in mode n) and the superblock size L. For mode-n MTTKRP, the number of independent superblocks is roughly $\frac{I_n}{L}$. If it is small (e.g., 2 for choa in mode 3), most superblocks cannot run simultaneously.

We introduce a widely used privatization parallel strategy [155, 156] to parallelize iterations, especially for irregularly-shaped tensors. Thread-local buffers are created locally to

keep the updates of the superblocks from distinct iterations, then they can be parallelized independently. Afterwards, an extra parallel reduction stage is used to get the final results. The privatization strategy is only used for MTTKRP in short tensor modes, i.e., small corresponding factor matrices, due to their disadvantage of direct parallelization. Besides, it consumes trivial extra memory. To better fit the dynamic workload of superblocks, we use dynamic OpenMP strategy for parallelism.

4.3.3 Parameter Guidance

Multiple parameters summarized in table 4.1 affect the performance of HICOO-MTTKRP and HICOO's storage size. We illustrate five critical parameters from our experiments and guide users to tune performance by narrowing down parameter space. Our program outputs suggestions to users if any parameter is out of a reasonable range. The first three are the parameters of the HICOO format, while the last two are choices among COO, CSF, and HICOO formats. Since our analyses in § 4.1, 4.2, 4.3 are for general tensors, these parameters could describe the features of both hypercubical and irregularly-shaped tensors which frequently occur in real applications.

Block Size. Block size B should be set to keep all the blocked matrices in fast cache, i.e., $NBR\beta_{\text{float}} \leq S_{cache}$. Users could compute the largest B for HICOO. This explains almost all HICOO-MTTKRPs obtain their best performance with $B = 128$ in our experiments, when $R = 16$.

Parallel Strategy. If the number of independent superblocks is too small (e.g., $< 4P$, P is the number of threads) to be parallelized efficiently and the iterations are much more than independent superblocks (e.g., $20\times$ larger in our experiments), we use privatization strategy to parallelize superblocks between iterations. This case generally occurs on irregularly-shaped tensors. For example, mode 3 of tensor choa is much smaller than the other two modes. Given a superblock size $L = 512$, the number of independent superblocks is only 2 in mode 3, while the number of iterations is as large as 22475. In this case, privatization

strategy is chosen for better performance.

Superblock Size. Superblock size L is important to parallel MTTKRP performance. It can be tuned to ensure reasonable amount of tasks for CPU threads. After fixing a parallel strategy, we can compute the number of parallel tasks (either independent superblocks or iterations) for different Ls. When this number is reasonable, e.g., between $[4P, 100P]$, this L is acceptable; otherwise, a smaller or larger L will be suggested.

Storage space. As explained in § 4.2, α_b determines the storage size of HICOO, smaller is better. The threshold of HICOO to achieve compressed storage than COO can be calculated from Equation (4.11). We suggest to first compare the value to its threshold (0.45 for 3D tensors), if α_b is larger than it, a HICOO representation is highly possible to use more storage than one COO and also CSF representation.

Performance. $\overline{c_b}$ is critical to HICOO-MTTKRP's memory traffic and thus its performance from table 4.2, larger is better. From our experiments, until $\overline{c_b} < 0.01$, HICOO-MTTKRP becomes slower than CSF-MTTKRP on tensors deli and nell1 by taking advantage of our parallel strategy. Besides, α_b is also taken into account because it affects the memory traffic of loading the input tensor, especially when the tensor is much larger than factor matrices. If α_b is larger and $\overline{c_b}$ is smaller than their thresholds, HICOO-MTTKRP may not behave well. Therefore, CSF-MTTKRP is favorable.

Note that these thresholds are obtained from the empirical tests on our platform. Users need to determine their own thresholds by trial-and-error runnings.

4.4 Experiments and Analyses

4.4.1 Experimental Setup

Platform The experiments are ran on a Linux server with Intel Xeon E7-4850 v3 multicore platform consisting 56 physical cores with 2.2 GHz frequency distributed on four sockets. It is a Haswell platform with 32 KiB L1 data cache and 1970 GiB memory. Code is written in C language with OpenMP directives, and is compiled by icc 18.0.2.

Dataset We use the sparse tensors, derived from real-world applications, that appear in Table 4.3, ordered by decreasing non-zero density separately for third- and fourth-order tensors. Most of these tensors are included in The Formidable Repository of Open Sparse Tensors and Tools (FROSTT) dataset (Refer to the details in [151]). The darpa (source IP-destination IP-time triples), fb-m, and fb-s (short for "freebase-music" and "freebase-sampled", entity-entity-relation triples) are from the dataset of HaTen2 [78], and choa is built from electronic health records (EHRs) of pediatric patients at Children's Healthcare of Atlanta (CHOA) [135].

Implementations We compare the performance of MTTKRP algorithms on multicore CPUs using COO and CSF formats[4]. COO-MTTKRP from ParTI! library [102] is a C implementation of the MTTKRP that appears in TENSOR TOOLBOX [16]. We use OpenMP with privatization method to obtain its highest possible performance at a cost of maybe large extra space to save the local copies. CSF-MTTKRP is from SPLATT v1.1.1 [157] which is regarded as the state-of-the-art MTTKRP and CPD library [156]. We configured SPLATT for its best performance, meaning the ALLMODE setting (storing all N CSF trees) and enabling the tiling option. All parallel programs are configured using "numactl" to interleave allocated memory system-wide, i.e., on all sockets.

Table 4.3: Description of sparse tensors.

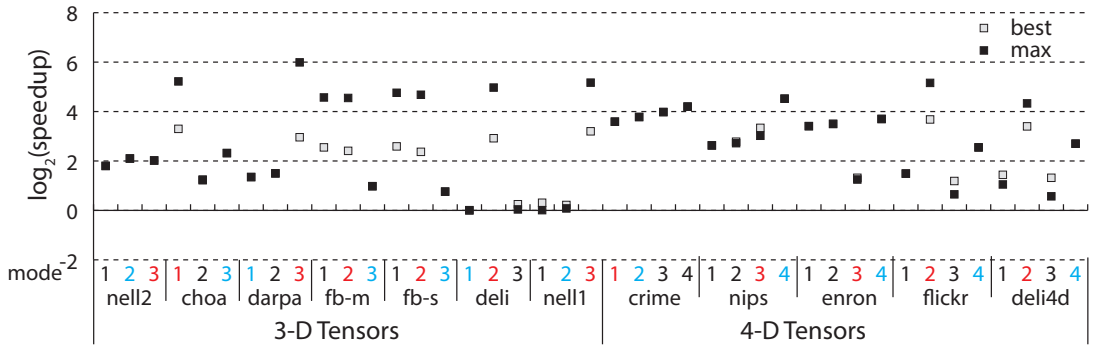
Tensors	Order	Dimensions	#Non-zeros	Density
nell2	3	$12K \times 9K \times 29K$	77M	2.4×10^{-5}
choa	3	$712K \times 10K \times 767$	27M	5.0×10^{-6}
darpa	3	$22K \times 22K \times 24M$	28M	2.4×10^{-9}
fb-m	3	$23M \times 23M \times 166$	100M	1.1×10^{-9}
fb-s	3	$39M \times 39M \times 532$	140M	1.7×10^{-10}
deli	3	$533K \times 17M \times 2.5M$	140M	6.1×10^{-12}
nell1	3	$2.9M \times 2.1M \times 25M$	144M	9.1×10^{-13}
crime	4	$6K \times 24 \times 77 \times 32$	5M	1.5×10^{-2}
nips	4	$2K \times 3K \times 14K \times 17$	3M	1.8×10^{-6}
enron	4	$6K \times 6K \times 244K \times 1K$	54M	5.5×10^{-9}
flickr	4	$320K \times 28M \times 1.6M \times 731$	113M	1.1×10^{-14}
deli4d	4	$533K \times 17M \times 2.5M \times 1K$	140M	4.3×10^{-15}

[4]F-COO is implemented only for GPUs.

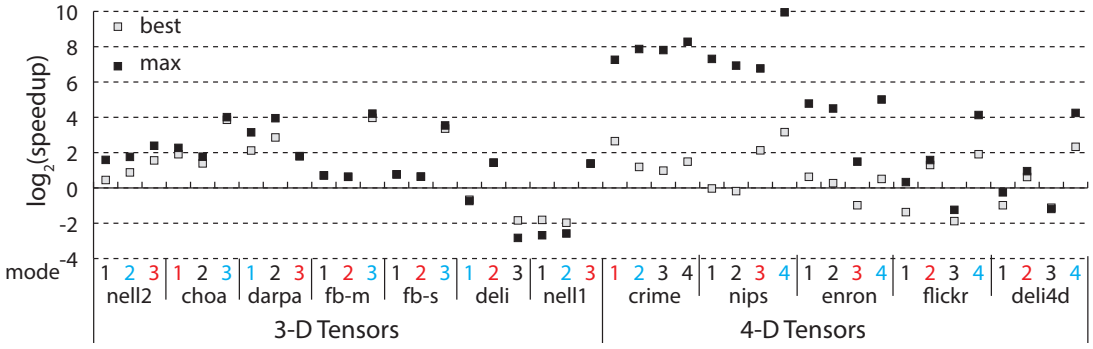
4.4.2 Overall Performance

The tensor rank R is set to 16 in our experiments. For all experiments, we use 32-bit unsigned integers for indices and pointers, which are enough for these tensors, and 32-bit single-precision floating-point values. Most experiments find $B = 128$ achieve the best performance, even though we can use up to 16 bits for large B, while L varies much from tensors to tensors. Generally, for a long tensor mode, direct parallelism is used; while for a very short mode, privatization method is chosen. All execution times are averaged over five iterations.

Figure 4.5 (a) and (b) show the speedup of parallel MTTKRP in a single mode based on HICOO over COO and CSF (ALLMODE setting) formats, respectively. We test up to 56 threads and show the performance obtained by the most threads ("max") and under the thread configuration with the highest performance ("best") separately. The y-axis shows the log_2(speedup) and x-axis gives all the cases of MTTKRPs, i.e., MTTKRPs in every mode on all tensors. A single parallel HICOO-MTTKRP achieves up to $63.5\times$ ($12.6\times$ on average) speedup over COO format and up to $991.4\times$ ($62.0\times$ on average) speedup over CSF format when using 56 threads. The extraordinary high speedup over CSF is achieved on tensors crime and nips, because CSF-MTTKRP scales poorly in their very short modes. Under the "best" thread configuration for all implementations, a single parallel HICOO-MTTKRP achieves up to $23.0\times$ ($6.8\times$ on average) speedup over COO and up to $15.6\times$ ($3.1\times$ on average) speedup over CSF. The highest performance of CSF-MTTKRP on tensors crime and nips is mostly obtained on 2 and 4 threads. In some tensor modes, only a black square can be recognized since the highest speedup is achieved by using all 56 threads. Mostly the "max" configuration gets higher or equal speedup than the "best" configuration because HICOO-MTTKRP is the most scalable in the three implementations (see Figure 4.7 below). However, in some cases when HICOO-MTTKRP is less well-scaled, the "best" configuration may get better speedup. In the following content, we only refer to the speedup achieves under the "best" thread configuration.



(a) Speedups of HICOO over COO



(b) Speedups of HICOO over CSF in ALLMODE setting

Figure 4.5: MTTKRP performance comparison with the "max" and "best" thread configurations. The longest modes and shortest modes of every tensor are marked in red and blue respectively.

As we mentioned, irregularly-shaped tensors have distinct performance behavior in different modes. We summarize the speedup separately for the longest (marked in red in Figure 4.5) and the shortest (marked in blue) modes of every tensor. HICOO-MTTKRP in the shortest mode achieves an average $6.7\times$ speedup over COO and $5.7\times$ speedup over CSF; while in the longest mode, HICOO-MTTKRP obtains an average $8.1\times$ speedup over COO and $2.8\times$ over CSF. HICOO is more advantageous in short modes compared to CSF, because of the less shape-sensitivity and privatization parallel strategy of HICOO. Compared to COO, HICOO shows a marginal win in long modes, because HICOO avoids the relatively expensive reduction stage of privatization method used in COO.

HICOO-MTTKRP is almost always faster than COO-MTTKRP due to its better data locality and smaller memory footprint. Compared to CSF-MTTKRP, the speedup of HICOO-MTTKRP is not stable because CSF-MTTKRP distinctly behaves on long and short modes, where long modes are more favorable. Some tensors, e.g., nell1 and deli, have almost all the blocks containing just a few non-zero entries, making them hypersparse ($\overline{c_b} \ll 1$). Thus, HICOO cannot reduce much memory traffic of MTTKRP (see table 4.2). Since our method flexibly chooses the two parallelization strategies, tensors fb-m and fb-s need only 249KB and 760KB extra space for the local matrix copies, trading that for much higher performance.

4.4.3 Optimization Breakdown

We show the advantages of HICOO by evaluating three factors separately, which are sorting, compression, and SIMD. The baseline is COO-MTTKRP with the COO representation sorted in a lexicographic mode order, then we use Z-order sorting to rearrange non-zeros but still run COO-MTTKRP, where we get 18% speedup on average. We convert tensors to HICOO by compressing indices, an extra 20% improvement is shown. Lastly, SIMD directives are applied to vectorize matrix rows, which gives an average of 22% speedup. With all these optimizations, HICOO-MTTKRP doubles the performance of COO-MTTKRP for

a single thread.

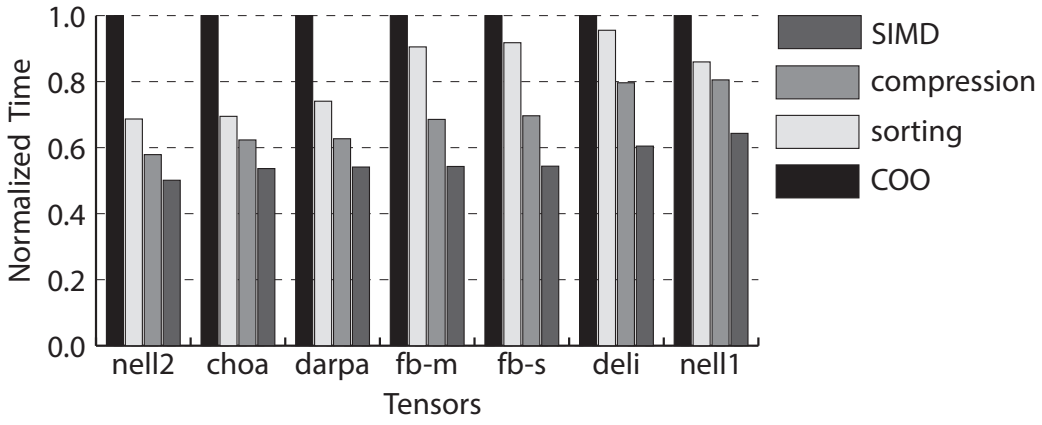
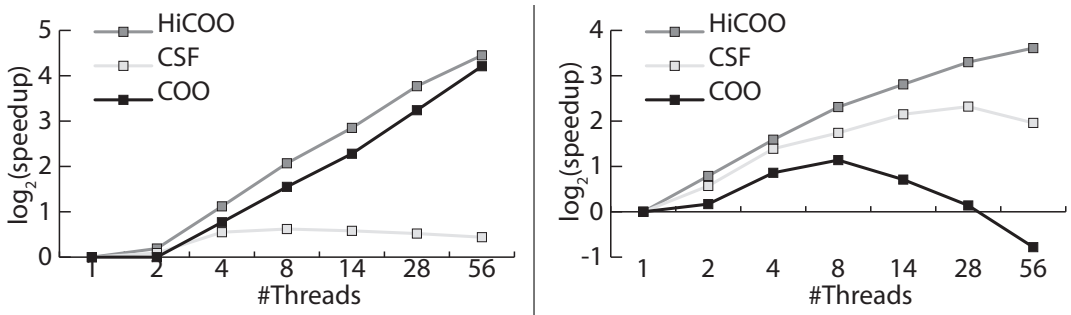


Figure 4.6: HICOO optimization breakdown on 3D tensors.

4.4.4 Thread Scalability

Figure 4.7 compares the thread scalability of COO, CSF, and HICOO MTTKRPs using tensors fb-s and choa representing the behavior in the shortest and longest modes of a tensor. Generally, CSF-MTTKRP scales poorly in short modes: 9 out of all 12 tensors achieve the best performance in their shortest modes with ≤ 8 threads; while only 2 tensors achieve the best performance in their longest modes with ≤ 8 threads. COO-MTTKRP does not scale well in long modes: 7 out of 12 tensors achieve the best performance in their longest mode with ≤ 8 threads. The x-axis shows the number of threads and the y-axis shows the log_2(speedup) over sequential MTTKRP. HICOO-MTTKRP scales better than COO and CSF in most cases despite the extra lptr and lschr structures needed to support superblock scheduling. CSF-MTTKRP has a potentially low parallel degree for short modes and may uses locks, while COO-MTTKRP always employs privatization with an expensive reduction stage for long modes, which affects its scalability. These observations verify the results in Section 4.4.2.



(a) tensor fb-s in mode 3 (shortest mode)

(b) tensor choa in mode 1 (longest mode)

Figure 4.7: Thread scalability of parallel COO, CSF, and HICOO MTTKRPs on two representative cases.

4.4.5 Storage Space

Table 4.4 compares the storage space of COO, CSF, F-COO, and HICOO formats, along with HICOO's compression rates over COO and its α_b values. We show two settings for CSF format, ALLMODE (using all CSF trees) and ONEMODE (using only one CSF tree), while all F-COO representations have to be stored. HICOO format uses less space than ALLMODE CSF and F-COO formats on all tensors, up to $2.5\times$ less storage than COO format, and comparable storage with ONEMODE CSF. HICOO consumes more space than COO on tensors deli, deli4d and nell1. As discussed in Section 4.3.3, the value of block ratio should satisfy $\alpha_b < 0.45$ (3D) or 0.5 (4D) for a tensor to take less space in HICOO than in COO. The α_b values of these three tensors are much larger than them, so it is not surprising their HICOO representations are larger. These observations suggest that a simple test, perhaps based on sampling, could be used to determine when to use COO versus HICOO from storage aspect, which supports our guidance in Section 4.3.3.

4.4.6 Superblock Size L

The choice of superblock size L affects the number of independent tasks available in HICOO-MTTKRP as discusses in Section 4.3.3. Figure 4.8 shows how the execution time of a HICOO-MTTKRP sequence varies with L on 3D tensors. The x-axis shows L values

Table 4.4: Sparse tensor space comparison in different formats.

Tensors	COO (MiB)	CSF (MiB) ALL	CSF (MiB) ONE	F-COO (MiB)	HICOO (MiB)	Compress Rate	α_b Values
choa	411	666	212	935	**192**	2.14	0.02
darpa	434	958	**218**	986	308	1.41	0.22
nell2	1150	1850	589	2667	**543**	2.12	0.02
fb-m	1480	3760	**1200**	3453	1420	1.04	0.42
fb-s	2080	5410	**1720**	4854	2100	0.99	0.46
deli	2090	4120	**1320**	4861	3490	0.60	0.99
nell1	2140	4430	**1210**	4981	3610	0.59	1.00
crime	102	176	**41**	328	**41**	2.49	0.00
nips	59	106	**24**	191	25	2.36	0.02
enron	1010	2030	**421**	3334	460	2.20	0.04
flickr	2100	4540	**1040**	6944	1740	1.21	0.36
deli4d	2610	7340	**1860**	8619	3540	0.74	0.80

on a log-scale, while the y-axis shows the execution times of parallel HICOO-MTTKRP normalized to that of $L = 2^8$ or 2^{10}. Tensors nell2 and choa show their best performance at $L = 2^{10}$ in figure 4.8(a). By contrast, MTTKRP times on the other tensors in figure 4.8(b) tend to decrease asymptotically with L, with some small variability beyond a certain point ($L = 2^{14}$). The main difference between these two groups is non-zero density, which is relatively high for tensors in figure 4.8(a) and low in figure 4.8(b). The guidance in Section 4.3.3 helps users to fast identify the optimal L.

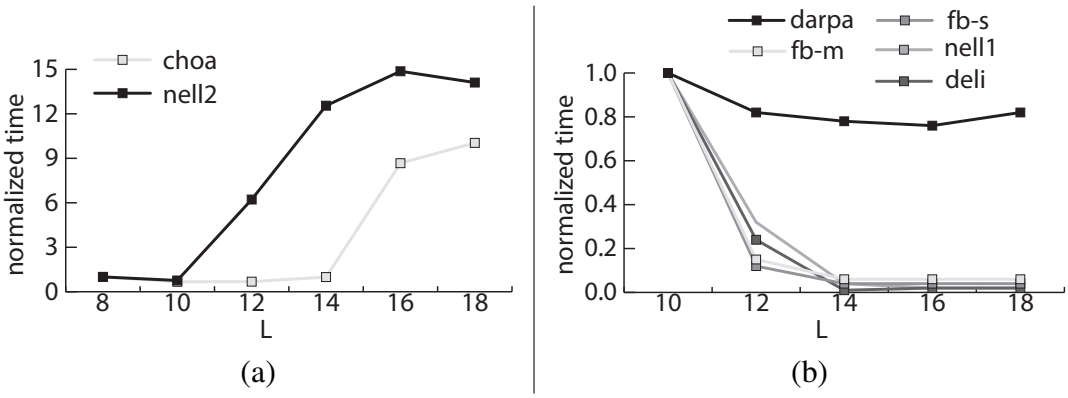


Figure 4.8: Superblock size L influences on HICOO-MTTKRP, x-axis shows $logL$ values, times are normalized to $L = 2^8$ or 2^{10}. Lower is better.

4.4.7 CPD

Figure 4.9 depicts a CANDECOMP/PARAFAC decomposition using alternating least squares algorithm (CP-ALS) [89] in all the three formats, where MTTKRP is the most expensive computational kernel. The speedup and compression rates are relative to CSF in ALLMODE setting, and "CSF-1" refers to CSF in ONEMODE setting. HICOO achieves the best speedup on most tensors with comparable compression rate to CSF ONEMODE. COO does not gain any advantages either from performance or storage aspect. HICOO-CPD outperforms $6.2\times$ over COO-CPD and $2.1\times$ over CSF-CPD on average.

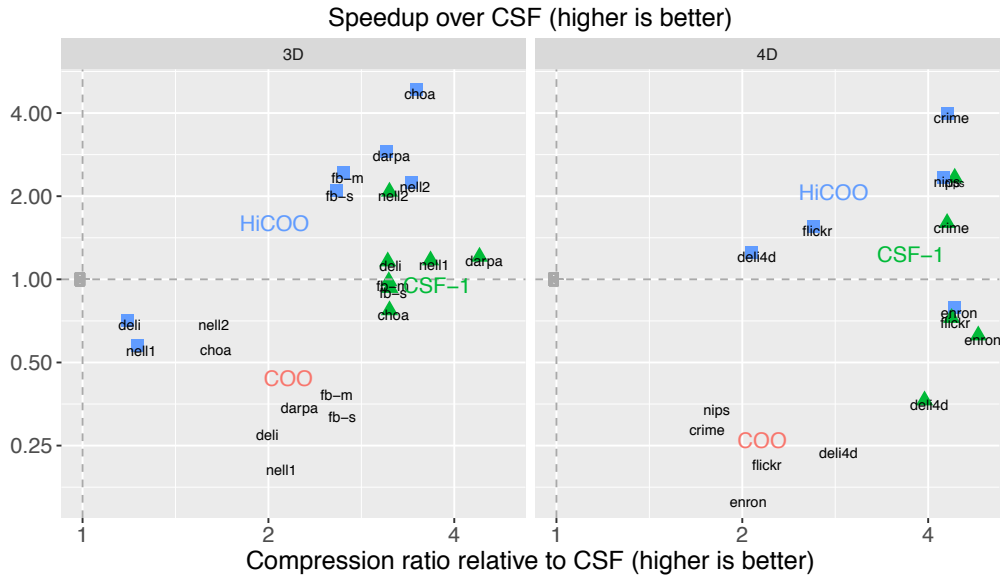


Figure 4.9: CPD time and tensor storage comparison relative to CSF in ALLMODE setting.

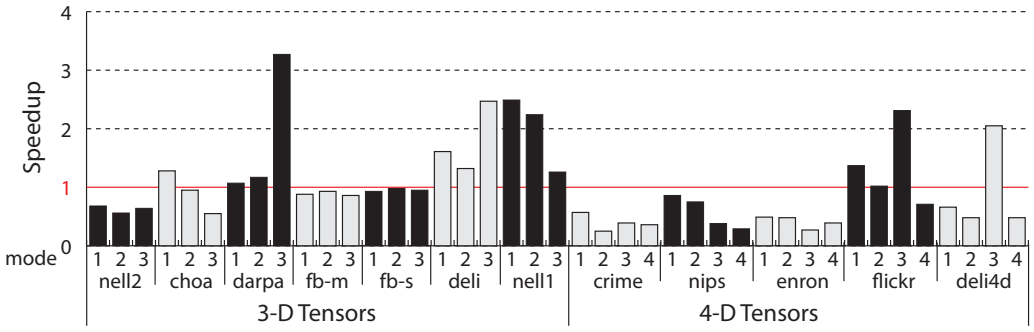


Figure 4.10: HICOO-MTTKRP speedup of KNL over Haswell.

4.4.8 Experiments on KNL

We also test HICOO-MTTKRP on a Intel Xeon Phi Processor 7250 ("Knights Landing") platform using the native mode. Figure 4.10 shows its speedup of KNL using 68 threads over Haswell using 56 threads. HICOO-MTTKRP on KNL achieves $0.25-3.27\times$ speedup, but Haswell behaves better in most cases. This experiment gives a proof-of-concept that other accelerators, e.g., GPUs, can also benefit from HICOO format with its good thread scalability.

4.5 Related Work

Study on improving the performance of sparse problems has a long history, for example, some optimization methods for sparse matrices are enlightening and applicable to tensors [12, 13, 31, 33, 43, 88, 105, 109, 110, 117, 120, 172, 173, 178, 182]. Optimizing CPD and the MTTKRP operation has been well-studied recently. TENSOR TOOLBOX [16] and Tensorlab [171] implemented MTTKRP in the most popular COO format through multiple sparse tensor-vector products using MATLAB. SPLATT [152, 156] algorithmically improved MTTKRP by factoring out inner multiplications and proposed a more compressed CSF format. According to the experiments in [156], it outperforms Tensor Toolbox, GigaTensor, and DFacTo (introduced below) on multicore CPUs, becoming the one achieving the highest performance by far. It has also applied to Intel Knights Landing many-core processor (KNL) recently [155]. GigaTensor [80] reformulated MTTKRP as a series of Hadamard products to utilize the massive parallelism of MapReduce. However, this algorithm is not suitable for multicore CPUs because of its high computational complexity. DFacTo [41] considered MTTKRP as a series of sparse matrix-vector multiplications for distributed systems, however, it requires explicit matricization which takes non-negligible time. HyperTensor [85] investigated fine- and also coarse-grained parallel algorithms for distributed systems, while its MTTKRP implementation is based on COO format. A more

recent work [107] proposed Flagged-COOrdinate (F-COO) format. As we state in Chapter 2 and § 4.1, CSF and F-COO formats are mode oriented, which can achieve compactness but not, simultaneously, neutral mode orientation. Some tensor formats are proposed for structural sparse tensors. "Mode-generic and mode-specific" [21] and semi-COOrdinate (sCOO) [103] formats are proposed for tensors with some modes dense.

Our work proposed a new compressed HICOO format for general sparse tensors, which does not favor one mode over the others and preserves the neutral mode orientation. HICOO format is proposed to explore an alternative approach of sparse tensor formats. Compared to the state-of-the-art CSF from SPLATT [156, 157], HICOO has the following advantages. First, HICOO could exploit data locality for all tensor dimensions while CSF's tree structure limits this mostly to the leaf level. Second, HICOO has more potential of its storage compression because of the avoidance of larger and indeterminate-length pointers compared to indices. We are pursuing more aggressive compression techniques, as proposed for sparse matrices [177], as part of our future work for HICOO. Third, from our parallel strategies and the experiments, HICOO-MTTKRP is less sensitive to irregularly-shaped tensors which frequently appear in real applications. Besides, our blocking strategy has similarities with cache tiling used in SPLATT, but is fine-grained and totally removes write conflicts. SPLATT uses coarse-grained parallelism which is quite similar to row parallelism in SpMV. This work is orthogonal to some optimization work for an MTTKRP sequence as a whole (Chapter 3), such as [86, 101], we will consider integrating them as our future work.

4.6 Summary

HICOO is a flexible, compact, and mode-neutral format for general sparse tensors. It derives from but also improves upon COO by compressing the indices in units of sparse tensor blocks, which compresses the storage while promoting data locality. Our multicore-parallel HICOO MTTKRP achieves remarkable speedups over COO and another state-of-the-art

format, compressed sparse fiber (CSF) formats. HiCOO uses up to $2.5\times$ less storage than COO format and comparable storage to one CSF representation. When used within CPD, we also observe speedups against COO- and CSF-based implementations.

CHAPTER 5
TENSOR REORDERING FOR HICOO

HICOO works better when tensor blocks are more dense, per the results of Section 4.4, because spatial and temporal locality improve with denser blocks. These observations raise a natural question, which is whether there is a reordering strategy that can increase the relative density of HICOO's blocks. By reordering, we mean a relabeling of the indices of one or more modes of the input tensor.

This chapter describes two such heuristic reordering schemes.[1] These two approaches arrange the non-zeros of a sparse tensor by ordering tensor indices along all modes, one after the other. One scheme uses a breadth-first search (BFS) heuristic based on the family of maximum cardinality search methods (and referred to as BFS-LIKE-MCS) [164]. It determines a permutation independently for each mode. The other scheme extends doubly lexical ordering techniques for matrices [112, 128] to tensors (called LEXI-ORDER). It lexicographically sorts every fiber, and iteratively repeats that process for all modes.

The main contributions of this chapter may be summarized as follows.

- We propose two tensor reordering approaches, BFS-LIKE-MCS and LEXI-ORDER, to enhance data locality by relabeling mode indices. (§ 5.1 and § 5.2)
- LEXI-ORDER improves the performance of parallel HICOO-, COO-, and CSF-MTTKRPs by up to $2.67\times$, $1.27\times$, and $1.72\times$ respectively. By comparing with another state-of-the-art reordering based on hypergraph partitioning [156], we find LEXI-ORDER achieves comparable performance on CSF-MTTKRP. While BFS-LIKE-MCS is proved to be less effective than LEXI-ORDER on these formats. (§ 5.3)
- By using the HICOO parameters (introduced in § 4.2) and the number of iterations

[1] The work described in this chapter is joint with Bora Uçar and Ümit Çatalyürek.

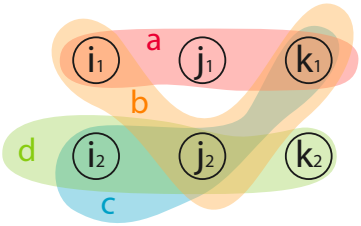
of LEXI-ORDER, we can determine a good tensor reordering and seek for a balance between performance and reordering overhead. (§ 5.3)

5.1 BFS-LIKE-MCS

BFS-LIKE-MCS is Breadth First Search (BFS)-like heuristic approach based on the maximum cardinality search family [164]. We first construct a hypergraph for a sparse tensor, where vertices are all tensor indices and hyperedges represent the non-zero entries. The weight of a vertex is the number of connections to it (calculated as shown below), and every hyperedge has unit weight (value 1). For a third-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ with M non-zeros, its hypergraph $H = (V, E)$ consists of $|V| = I_1 + I_2 + I_3$ vertices and $|E| = M$ hyperedges. A non-zero entry $x_{i_1 i_2 i_3}$ connects the 3 vertices i_1, i_2, i_3. Figure 5.1 shows an example of the hypergraph for a sparse tensor. Vertices are blank circles, and hyperedges are represented by grouping vertices.

i	j	k	val
i_1	j_1	k_1	a
i_1	j_2	k_1	b
i_2	j_2	k_1	c
i_2	j_2	k_2	d

(a) Tensor



(b) Hypergraph

Figure 5.1: A hypergraph example of a sparse tensor.

For an Nth-order sparse tensor, we need to find the permutations for N modes. We determine a permutation for a given mode n (perm_n) of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ as follows. Suppose some of the indices of mode n are already ordered, BFS-LIKE-MCS picks the next to-be-ordered index as the one with the strongest connection to the currently ordered index set. In the case of ties, it selects the index with the smallest number of non-zeros in the corresponding sub-tensor. Intuitively, in the hypergraph of a sparse tensor, a stronger connection represents more common indices in modes other than n, which potentially means more data reuse of matrices.

This process is implemented by maintaining a max-heap (H_n) for all mode-n indices (e.g., i_n). The primary key of each is the number of connections to the currently ordered indices and the secondary key is the minus of the number of non-zeros in its corresponding $(N-1)$th-order sub-tensor (e.g., $\mathcal{X}(:,\ldots,:,i_n,:,\ldots,:)$). Initially, H_n is constructed according to the secondary keys of mode-n indices. For each mode-n index (e.g., i_n) of this max-heap, BFS-LIKE-MCS traverses all connected tensor indices in the modes except n, calculates the number of connections of mode-n indices, and then updates the heap (H_n) using their new primary keys.

Algorithm 8 BFS-LIKE-MCS ordering based on maximum cardinality search for a given mode.

Input: An Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, hypergraph $G = (V, E)$ with vertex weights $\mathbf{w}$, mode n;
Output: Permutation $perm_n$;

Initialize $\mathbf{w}_p$ to zeros.
Initialize $\mathbf{w}_s$ to the minus of number of non-zeros of each corresponding sub-tensor.
Build max-heap H_n for mode-n indices with $\mathbf{w}_p$ and $\mathbf{w}_s$ as the primary and secondary keys respectively.
Initialize $\mathbf{m}_V$ to zeros.

for $i_n = 1, \ldots, I_n$ **do**
    $v_n^{(0)}$ = GetHeapMax(H_n);
    $perm_n(v_n^{(0)}) = i_n$;
    $\mathbf{m}_V(v_n^{(0)}) = 1$;
    for $e_n \in$ hyperedges of $v_n^{(0)}$ **do**
        for $v \in e_n$ and v is not in mode n and $\mathbf{m}_V(v) == 0$ **do**
            $\mathbf{m}_V(v) = 1$;
            for $e \in$ hyperedges of v and $e \neq e_n$ **do**
                $v_n =$ vertex in mode n of e;
                if inHeap(v_n) **then**
                    $\mathbf{w}_p(v_n) + +$;
                    heapUpdateKey($H_n, \mathbf{w}_p(v_n)$);
return $perm_n$;

Algorithm 8 makes BFS-LIKE-MCS more precise. Let m_V denote a size-$|V|$ array that tracks whether a vertex has been visited. They are initialized to zeros (unvisited) and changed to 1 once after being visited. We record a mode-n index $v_n^{(0)}$, obtained from the max-heap H_n, to the permutation array perm_n. The algorithm visits all its hyperedges (i.e.,

all non-zeros with i_n, Line 9) and their connected and unvisited vertices from modes other than n (Line 10). For these vertices, we again visit their hyperedges (e in Line 12) and then check if the connected vertices in mode n (v_n) are in the remaining heap (Line 14). If so, the primary key (connectivity) of v_n is increased by 1 and the max-heap (H_n) is updated.

The BFS-LIKE-MCS procedure is heuristic for two reasons. First, the hypergraph does not reflect the exact memory access pattern of an MTTKRP. BFS-LIKE-MCS treats the contribution from the indices of all modes except n equally to the connectivity, thus the connectivity might not match the actual data reuse. Secondly, it uses a greedy strategy to determine the next-level vertices, which could miss the optimal global orderings.

5.2 LEXI-ORDER

LEXI-ORDER is an extension of doubly lexical ordering of matrices [112, 128] to tensors. It lexicographically sorts every fiber, iteratively for all modes.

A lexicographic ordering of an integer vector is the standard dictionary ordering of its elements. The comparison operation between two vectors is defined differently [128]. Given two equal-length vectors, $\mathbf{x}$ and $\mathbf{y}$, we say $\mathbf{x} \leq \mathbf{y}$ iff either (i) all elements are the same, i.e., $\mathbf{x} = \mathbf{y}$; or (ii) there exists an index j such that $\mathbf{x}(j) < \mathbf{y}(j)$ and $\mathbf{x}(i) = \mathbf{y}(i)$ for all $0 \leq i < j$. For example, figure 5.2(a) shows the vector comparison of two zero-one vectors, $\mathbf{x} \equiv (1, 0, 1, 0)$ and $\mathbf{y} \equiv (1, 1, 0, 0)$. $\mathbf{x} \leq \mathbf{y}$ because $\mathbf{x}(1) < \mathbf{y}(1)$ and $\mathbf{x}(i) = \mathbf{y}(i)$ for all $0 \leq i < 1$. Give another example by considering the first two row vectors of figure 5.2(b), where $\mathbf{x} \equiv (0, 1, 0, 0)$ and $\mathbf{y} \equiv (0, 1, 0, 1)$. Then $\mathbf{x} \leq \mathbf{y}$ because $\mathbf{x}(3) < \mathbf{y}(3)$ and $\mathbf{x}(i) = \mathbf{y}(i)$ for all $0 \leq i < 3$. To obtain a lexicographic ordering of a set of vectors, one may sort the vectors using this vector $\leq$ operator.

A *doubly lexical ordering* of a matrix is an ordering of the rows and the columns of the matrix so that both the row vectors and the column vectors are in non-increasing lexicographic order. We specify to read a row vector from left to right and a column vector from top to bottom. Figure 5.2(b) shows a doubly lexical ordering of a zero-one example

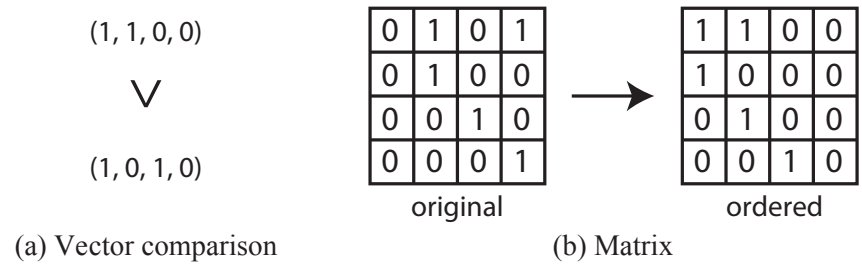


Figure 5.2: A doubly lexical ordering of a zero-one matrix and its vector comparison operation.

matrix. The row vectors are in non-increasing lexicographic order from top to bottom, and the column vectors are in non-increasing lexicographic order from left to right. Assume the ordered matrix in figure 5.2(b) is $\mathbf{A}$, then rows $\mathbf{a}(1,:) \geq \mathbf{a}(2,:) \geq \mathbf{a}(3,:) \geq \mathbf{a}(4,:)$ and columns $\mathbf{a}(:,1) \geq \mathbf{a}(:,2) \geq \mathbf{a}(:,3) \geq \mathbf{a}(:,4)$. Even from this simple example, the ordered matrix has better block locality than the origin one. Doubly lexical ordering has a number of applications, including the efficient recognition of totally balanced matrices, subtree matrices and plaid matrices and (strongly) chordal graphs.

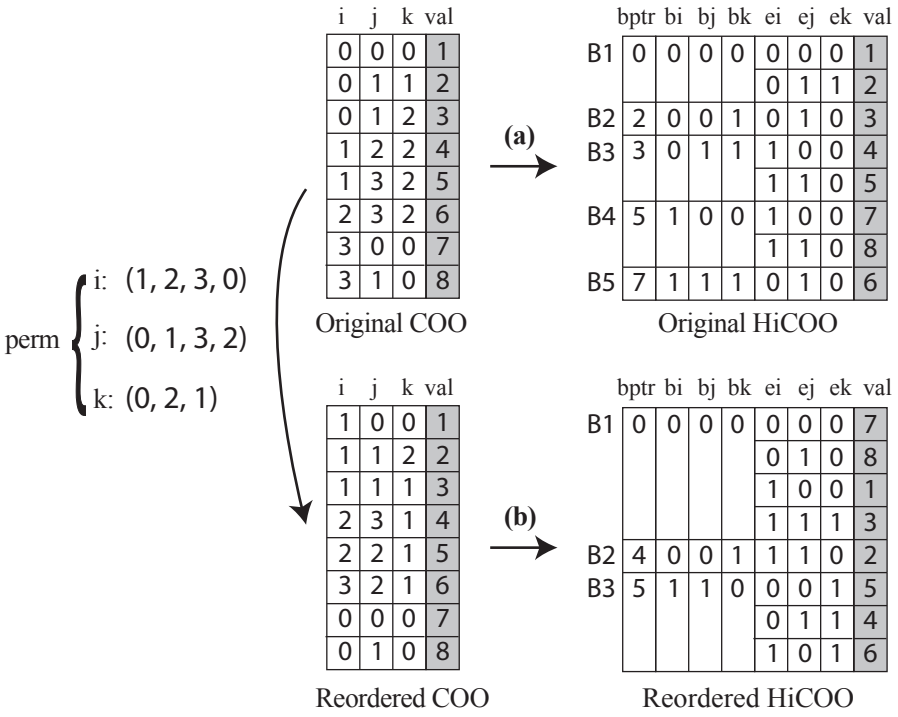


Figure 5.3: Comparison of HICOO representations before and after LEXI-ORDER— a good example.

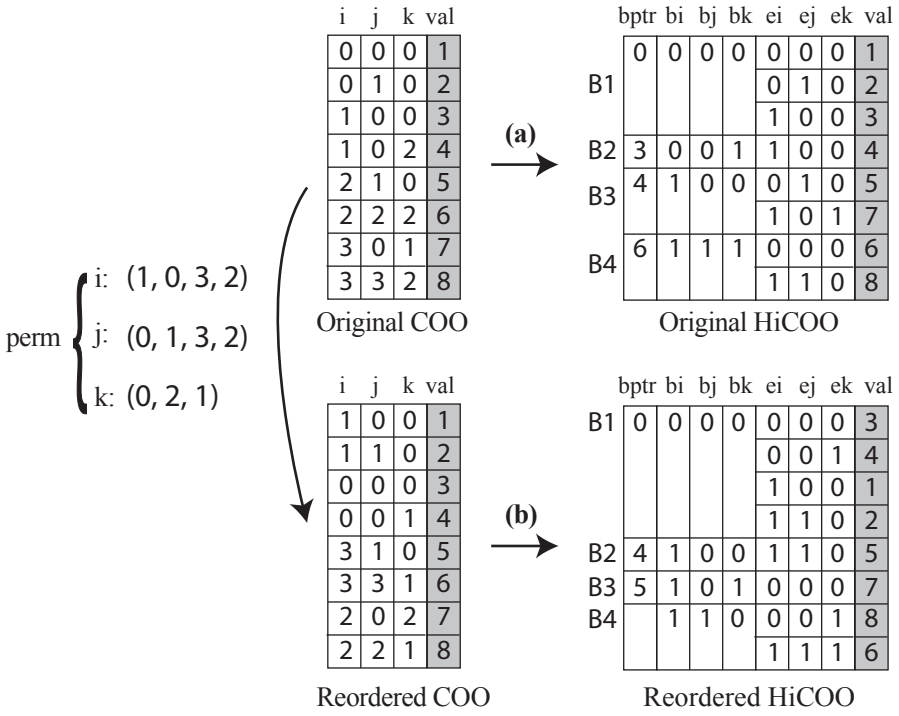


Figure 5.4: Comparison of HICOO representations before and after LEXI-ORDER— a fair example of figure 4.2.

Like BFS-LIKE-MCS approach, LEXI-ORDER also finds the permutations for N modes of an Nth-order sparse tensor. Figure 5.3 illustrates the effect of LEXI-ORDER on an example $4 \times 4 \times 3$ sparse tensor. The original tensor converts to a HICOO representation with block size $B = 2$ consisting of 5 tensor blocks, with maximum 2 non-zeros per block. After reordering with LEXI-ORDER, the new HICOO has 3 non-zero blocks with up to 4 non-zeros per block. Thus, the blocks are more dense, which should exhibit better locality behavior. However, this reordering scheme is heuristic. For example, consider figure 4.2, we draw another HICOO representation after reordering in figure 5.4. Applying LEXI-ORDER would yield a reordered HICOO representation with 4 tensor blocks, which is the same as the input ordering, although the maximum number of non-zeros per block would increase to 4. For this kind of tensor, LEXI-ORDER may not show a big advantage.

The basic idea of LEXI-ORDER is to determine the permutation of each tensor mode independently, where for each mode it performs a doubly lexical ordering on the matricized

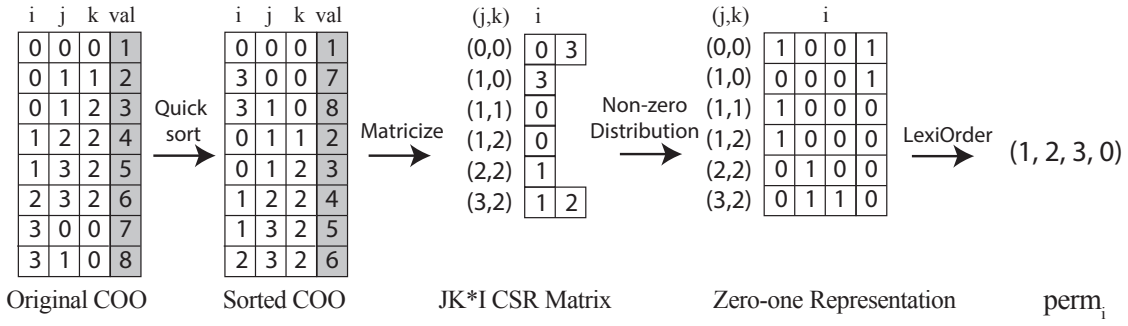


Figure 5.5: The steps of LEXI-ORDER (Algorithm 9) illustrated.

tensor [112, 128]. We start with a logical zero-one matrix to represent the non-zero distribution of a matricized tensor, and stored using Compressed Sparse Row (CSR) format. We then use a variation of the partition refinement algorithm to compute its doubly lexical ordering [128]. The permutations can be obtained by tracking this ordering process.

The precise algorithm appears in Algorithm 9, which we also illustrate in figure 5.5 when applied to mode 1. Given a mode n, LEXI-ORDER first sorts the non-zeros of the tensor using quicksort with the comparison function `coordCmp`. This comparison function does a lexicographic comparison of the all-but-mode-n indices. This sorting of the tensor $\mathcal{X}$ is the same as sorting the matricized tensor $\mathbf{X}_{(n)}$ by rows, where mode n is the column dimension and the remaining modes constitute the rows. In figure 5.5, the sorting step orders the COO entries by (j, k) tuples, which then serve as the row indices of the matricized CSR representation of $\mathbf{X}_{(1)}$. Since the sorted tensor is ordered by increasing rows of $\mathbf{X}_{(n)}$, we can easily matricize tensor $\mathcal{X}$ into a CSR representation by partitioning row segments (Lines 3–7). Then, we use a similar partition refinement algorithm `lexiOrder` to the Paige and Tarjan scheme [128] for every row of the matrix $\mathbf{X}_{(n)}$. We use zero-one row vectors in figure 5.5 to illustrate its non-zero distribution, which could seemlessly call `lexiOrder` function.

The doubly lexical ordering algorithms for matrices, which were first explored by Lubiw [112] and then improved by Paige and Tarjan [128], are "direct" ordering methods with a non-linear runtime of $\mathcal{O}(M \log(I + J) + J)$ and $\mathcal{O}(M + I + J)$ space, for an $I \times J$

matrix with M non-zeros. We propose an iterative algorithm (`lexiOrder`), where each iteration has a linear runtime complexity $\mathcal{O}(M + I + J)$ and uses $\mathcal{O}(J)$ space. Compared to the prior doubly lexical ordering methods, our `lexiOrder` is sufficient for our needs in that we do not need a fully ordered tensor, thereby making our approach faster and simpler to implement. And we observe that in practice that not only is every iteration at worst linear, but usually only a small number of iterations are needed. The bottleneck in `lexiOrder` is the sorting phase (`quickSort`).

Algorithm 9 Multiply lexical ordering for a given mode.

Input: An Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, mode n;
Output: Permutation perm_n;

▷ Sort all non-zeros along with all but mode n.
quickSort($\mathcal{X}$, coordCmp);
▷ Matricize $\mathcal{X}$ to $\mathbf{X}_{(n)}$.
$r =$ compose (inds $([-n]), 1)$;
for $m = 1, \ldots, M$ **do**
    $c = \text{inds}(n, m)$; ▷ Column index of $\mathbf{X}_{(n)}$
    if coordCmp($\mathcal{X}, m, m-1$) == 1 **then**
        $r =$ compose (inds $([-n]), m)$; ▷ Row index of $\mathbf{X}_{(n)}$
    $\mathbf{X}_{(n)}(r, c) = \text{val}(m)$;
▷ Use a variation of partition refinement in [128]
$\text{perm}_n =$ lexiOrder($\mathbf{X}_{(n)}$);
return perm_n;
▷ Comparison function for two indices of $\mathcal{X}$
Function: coordCmp($\mathcal{X}, m_1, m_2$)
for $n' = 1, \ldots, N$ **do**
    if $n'! = n$ **then**
        if $m_1(n') < m_2(n')$ **then**
            return -1; ▷ Entry $m_1 <$ entry m_2
        if $m_1(n') > m_2(n')$ **then**
            return 1; ▷ Entry $m_1 >$ entry m_2
return 0; ▷ Entry $m_1 =$ entry m_2

5.3 Evaluation

5.3.1 Experimental Setup

Platform We tested these schemes experimentally on a Linux-based Intel Xeon E5-2698 v3 multicore server platform with 32 physical cores distributed on two sockets, each with 2.3 GHz frequency. The processor microarchitecture is Haswell, having 32 KiB L1 data cache and 128 GiB memory. The code artifact is written in the C language using OpenMP parallelization, and was compiled using icc 18.0.1.

Dataset We use the sparse tensors, derived from real-world applications, that appear in Table 5.1, ordered by decreasing non-zero density separately for third- and fourth-order tensors. Most of these tensors are included in The Formidable Repository of Open Sparse Tensors and Tools (FROSTT) dataset [151]. The darpa (source IP-destination IP-time triples), fb-m, and fb-s (short for "freebase-music" and "freebase-sampled", entity-entity-relation triples) are from the dataset of HaTen2 [78], and choa is built from electronic health records (EHRs) of pediatric patients at Children's Healthcare of Atlanta (CHOA) [135].

Table 5.1: Description of sparse tensors.

Tensors	Order	Dimensions	#Non-zeros	Density
vast	3	$165K \times 11K \times 2$	26M	6.9×10^{-3}
nell2	3	$12K \times 9K \times 29K$	77M	2.4×10^{-5}
choa	3	$712K \times 10K \times 767$	27M	5.0×10^{-6}
darpa	3	$22K \times 22K \times 24M$	28M	2.4×10^{-9}
fb-m	3	$23M \times 23M \times 166$	100M	1.1×10^{-9}
fb-s	3	$39M \times 39M \times 532$	140M	1.7×10^{-10}
flickr	3	$320K \times 28M \times 1.6M$	113M	7.8×10^{-12}
deli	3	$533K \times 17M \times 2.5M$	140M	6.1×10^{-12}
nell1	3	$2.9M \times 2.1M \times 25M$	144M	9.1×10^{-13}
crime	4	$6K \times 24 \times 77 \times 32$	5M	1.5×10^{-2}
uber	4	$183 \times 24 \times 1140 \times 1717$	3M	3.9×10^{-4}
nips	4	$2K \times 3K \times 14K \times 17$	3M	1.8×10^{-6}
enron	4	$6K \times 6K \times 244K \times 1K$	54M	5.5×10^{-9}
flickr4d	4	$320K \times 28M \times 1.6M \times 731$	113M	1.1×10^{-14}
deli4d	4	$533K \times 17M \times 2.5M \times 1K$	140M	4.3×10^{-15}

5.3.2 HICOO-MTTKRP with Reordering

Figure 5.6 shows the speedup of the two reordering methods on sequential HICOO-MTTKRP. We use the total execution time of MTTKRPs on all modes for each tensor. The speedup is the ratio of the execution time of no-reordering HICOO-MTTKRP over the one using reordering methods. We show the best configurations (the number of iterations and the block size B) for LEXI-ORDER to get the highest MTTKRP performance. Similar to the experiments in § 4.4, block size $B = 128$ achieves the best results in most cases. The effect of the number of iterations will be shown in figure 5.13.

LEXI-ORDER reordering gets $0.97 - 2.35\times$ speedup ($1.48\times$ on average) for sequential HICOO-MTTKRP; while BFS-LIKE-MCS reordering does not accelerate much on most tensors except deli and nell1, with $1.07\times$ speedup on average. LEXI-ORDER and BFS-LIKE-MCS do not behave as well on fourth-order tensors as on third-order tensors. Tensor deli4d is constructed from the same data with deli, with an extra short mode (table 5.1). LEXI-ORDER obtains $2.29\times$ speedup on deli while only $1.14\times$ speedup on deli4d. The same phenomenon is also observed on tensors flickr and flickr4d. This phenomenon indicates that it is harder to get good block locality on higher-order tensors, which will be justified in table 5.2.

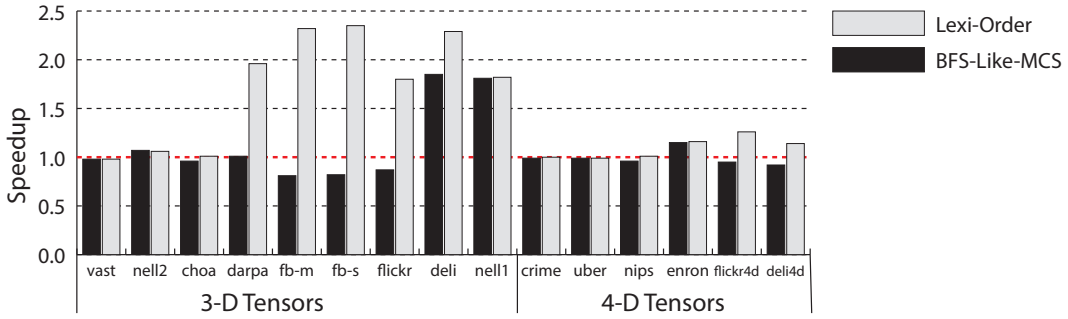


Figure 5.6: Reordered sequential HICOO-MTTKRP speedup.

The speedup of the two reordering methods on multicore parallel HICOO-MTTKRP appears in figure 5.7. We report the result for the best configuration, considering the number of iterations, the superblock size L, and the block size B. The block size $B = 128$ achieves

the best results in most cases. The best superblock size L can change under reordering (see table 5.2). Overall, LEXI-ORDER results in $0.65 - 2.67\times$ speedup ($1.23\times$ on average) for a parallel HICOO-MTTKRP; while BFS-LIKE-MCS reordering gets $0.88 - 1.04\times$ speedup ($0.98\times$ on average). Thus, the benefit from reordering using either LEXI-ORDER or BFS-LIKE-MCS is less than in the sequential case. This observation may indicate that thread scheduling may overcome some of the overhead from poor data locality.

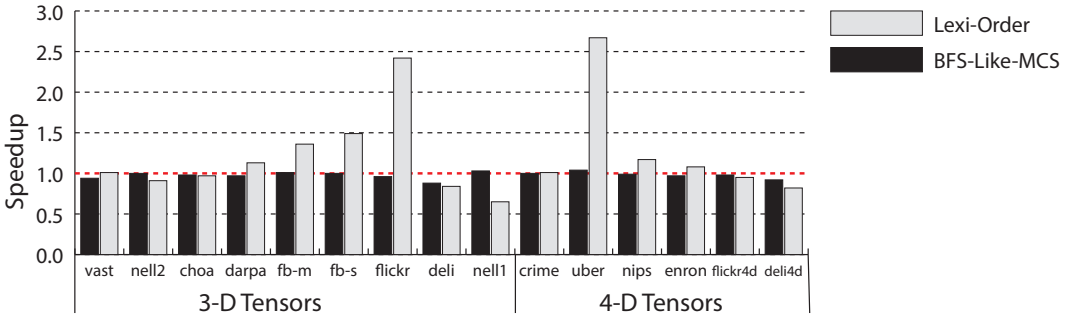


Figure 5.7: Reordered parallel HICOO-MTTKRP speedup.

5.3.3 Other Formats with Reordering

We also ran the MTTKRP experiments, with and without reordering, when storing the tensor in COO and CSF formats.

COO-MTTKRP

We show the effect of the two reordering approaches on sequential and parallel COO-MTTKRP from PARTI! [102] in figure 5.8 and 5.9. This COO-MTTKRP is implemented in C and OpenMP parallelized using privatization, using the same algorithm as TENSOR TOOLBOX [16]. Observe that LEXI-ORDER improves sequential COO-MTTKRP performance by $0.85 - 2.41\times$ ($1.42\times$ on average), while BFS-LIKE-MCS does not yield much improvement. Also, LEXI-ORDER improves parallel COO-MTTKRP performance by $0.94 - 1.27\times$ ($1.04\times$ on average), while BFS-LIKE-MCS improves by $0.64 - 1.40\times$ ($1.02\times$ on average). Thus, the qualitative effects of reordering on performance is the same

for COO-MTTKRP as it is for HICOO-MTTKRP, but with less quantitative improvement in the parallel cases.

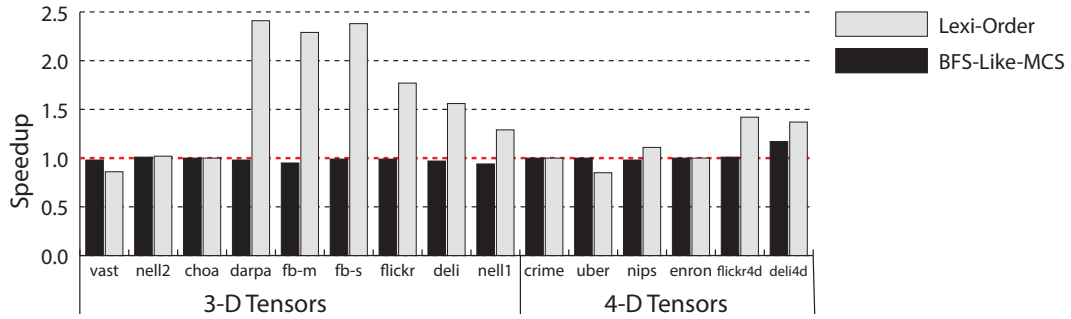


Figure 5.8: Reordered sequential COO-MTTKRP speedup over an unordered implementation.

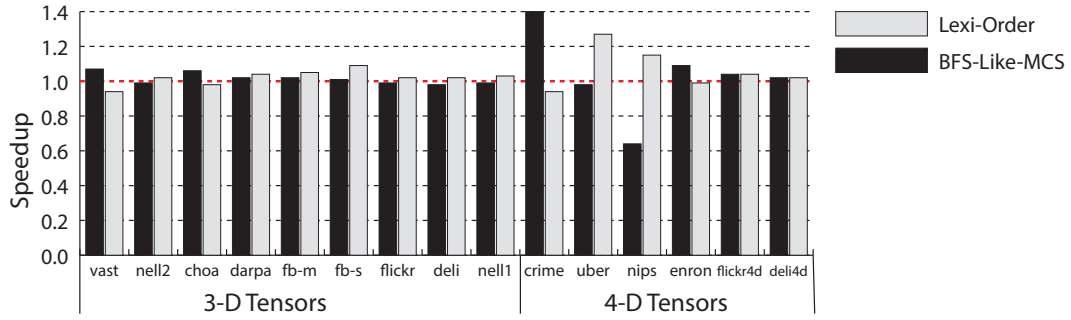


Figure 5.9: Reordered parallel COO-MTTKRP speedup over an unordered implementation.

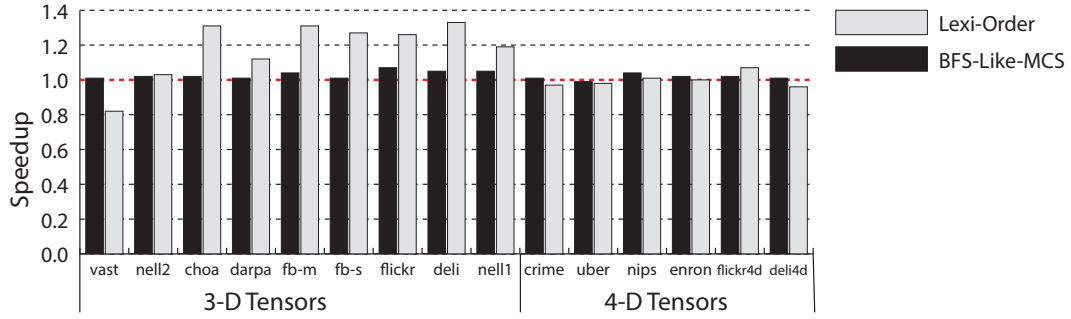


Figure 5.10: Reordered sequential CSF-MTTKRP speedup over an unordered implementation.

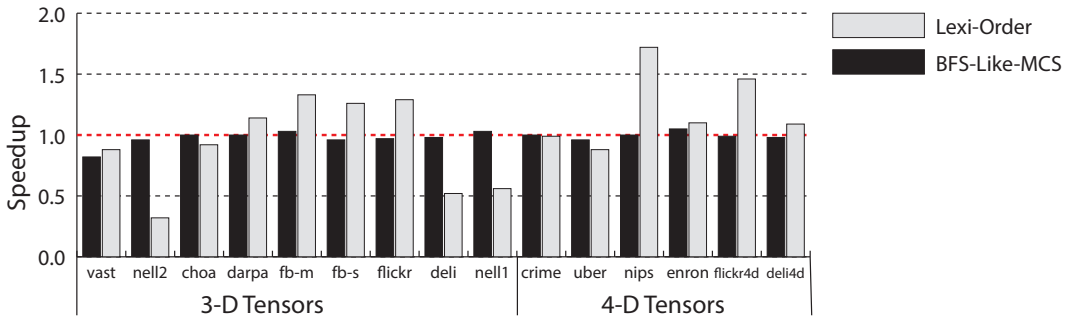


Figure 5.11: Reordered parallel CSF-MTTKRP speedup over an unordered implementation.

CSF-MTTKRP

We test the effect of the two reordering approaches on sequential and parallel CSF-MTTKRP from SPLATT v1.1.1 [157] in figure 5.10 and 5.11. CSF-MTTKRP is set to use only one CSF representation (ONEMODE) for MTTKRPs in all modes and with tiling option on. Our tests using all CSF representations (ALLMODE) show quite similar results to the ONEMODE setting. LEXI-ORDER improves sequential CSF-MTTKRP performance by $0.82 - 1.33\times$ ($1.11\times$ on average) and speedups parallel CSF-MTTKRP performance by $0.32 - 1.72\times$ ($1.03\times$ on average). The BFS-LIKE-MCS method does not improve performance much for either the sequential and parallel CSF-MTTKRPs. These two reordering approaches, especially LEXI-ORDER, do not perform as well on CSF as on HICOO and COO formats.

Format Comparison

Figure 5.12 compares the parallel performance of COO-, CSF-, and HICOO-MTTKRPs with the LEXI-ORDER approach. The performance is shown by the execution time of each implementation and normalized to the time of COO-MTTKRP for each tensor. HICOO still achieves the best performance as tested in § 4.4, which is $1.1 - 27.0\times$ ($7.1\times$ on average) faster than COO and $0.8 - 9.5\times$ ($3.5\times$ on average) faster than CSF. Because of CSF's poor thread scalability (discussed in § 4.4), CSF-MTTKRP runs even slower than COO-MTTKRP on some fourth-order tensors. Overall, HICOO format has been testified to be the most preferable one with or without tensor reordering.

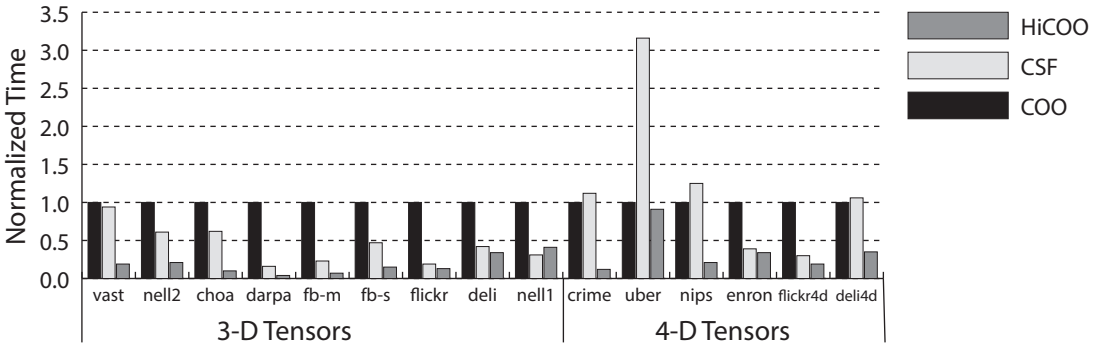


Figure 5.12: Format comparison with LEXI-ORDER on parallel MTTKRPs.

5.3.4 Reordering Methods Comparison

From the above experiments on HICOO, COO, and CSF formats, BFS-LIKE-MCS does not get much performance enhancement on all formats and almost all tensors. This is mainly because BFS-LIKE-MCS only takes connectivity into account and does not consider much local non-zero distribution or memory access pattern. Compared to the reordering method used in SPLATT [156], by setting ALLMODE (identical to Shaden et al.'s work [156]) to CSF-MTTKRP, BFS-LIKE-MCS gets 0.95, 0.99, and $1.00\times$ speedup on tensors nell2, nell1, and deli respectively. By contrast, the speedups from hypergraph partitioning [156] on these three tensors are 1.06, 1.12, and $1.24\times$, respectively. However, hypergraph partitioning is mode-dependent, which means N different permutations are generated for a Nth-order tensor. Thus, one might expect it to perform better than BFS-LIKE-MCS, which only determines one permutation. LEXI-ORDER behaves comparably with hypergraph partitioning, obtaining 1.03, 1.16, and $1.22\times$ speedup on the three tensors respectively. Moreover, LEXI-ORDER is not mode dependent and only needs one permutation.

5.3.5 Effect of the Number of Iterations in LEXI-ORDER

Since LEXI-ORDER improves the ordering iteratively, we evaluated the effects of the number of iterations on HICOO-MTTKRP performance. The results appear in figure 5.13.

When using 5 iterations, LEXI-ORDER obtains the shortest HICOO-MTTKRP time on the most tensors (7 instances). However, using 3 iterations only increases the running time on an average by about 1.8% over using 5 iterations. Moreover, according to figure 5.14, the reordering time of 3 iterations is only 60% that of 5 iterations on average. Therefore, a rough empirical guideline is that as few as 3 iterations may be sufficient to obtain an acceptable performance.

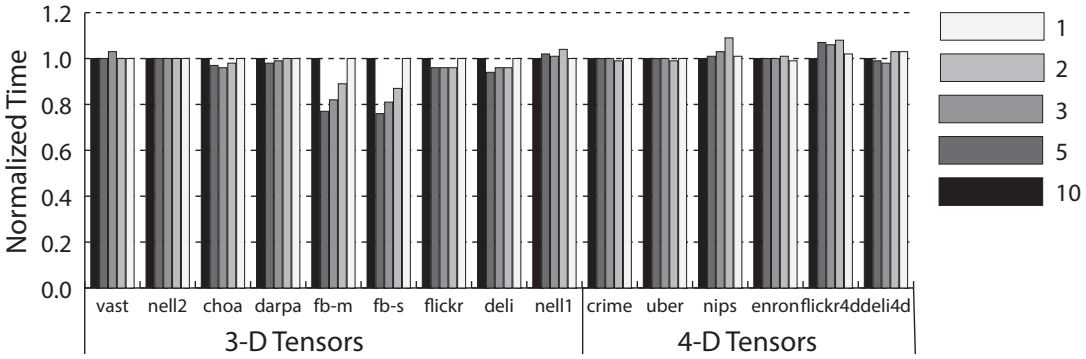


Figure 5.13: Sequential HICOO-MTTKRP behavior varying the number of iterations.

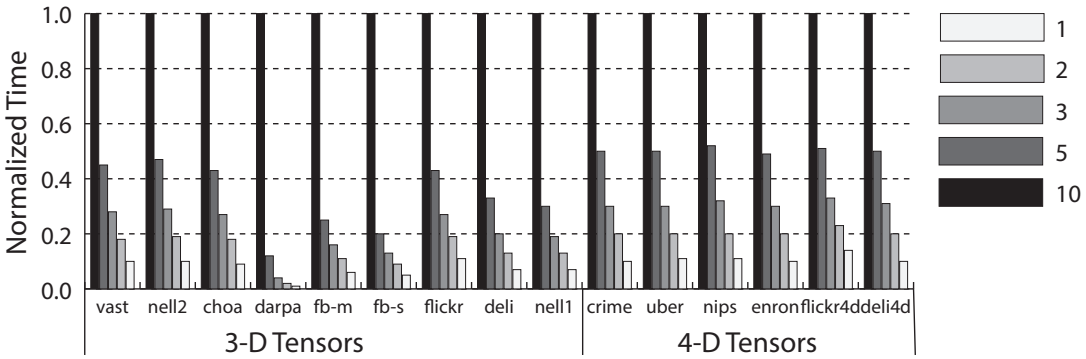


Figure 5.14: The reordering time of different numbers of iterations.

5.3.6 HICOO Parameters

We use the three most critical parameters introduced in § 4.2 and § 4.3.3, the block ratio (α_b), the average slice size per tensor block ($\overline{c_b}$), and the superblock size L. From § 4.3.3, smaller α_b and larger $\overline{c_b}$ are better for HICOO-MTTKRP performance. The best value of L depends on the tensor size and non-zero distribution, and affects the number of tasks

available to the superblock scheduler. Table 5.2 lists the three parameter values for all tensors before and after LEXI-ORDER, and the HICOO-MTTKRP speedup using LEXI-ORDER over no-ordering cases. Generally, when both parameters α_b and $\overline{c_b}$ are improved, we see a good speedup using LEXI-ORDER. Also, the optimal L values differ before and after reordering, particularly on the tensors nell2, fb-m, fb-s, flickr4d, and deli4d. It appears necessary to re-tune L after tensor reordering.

Table 5.2: HICOO parameters change before and after LEXI-ORDER reordering.

Tensors	No reordering			LEXI-ORDER			Speedup	
	α_b	$\overline{c_b}$	L	α_b	$\overline{c_b}$	L	seq	omp
vast	0.003	2.210	8	0.004	1.562	8	0.98	1.01
nell2	0.020	0.302	10	0.008	0.074	9	1.06	0.91
choa	0.023	0.070	10	0.016	0.056	10	1.01	0.97
darpa	0.217	0.016	15	0.018	0.113	15	1.96	1.13
fb-m	0.416	0.011	18	0.086	0.021	16	2.32	1.36
fb-s	0.456	0.010	18	0.099	0.020	16	2.35	1.49
flickr	0.358	0.014	13	0.097	0.025	13	1.80	2.42
deli	0.988	0.008	16	0.501	0.010	16	2.29	0.84
nell1	0.998	0.008	18	0.744	0.009	18	1.82	0.65
crime	0.000	666.892	4	0.000	654.686	4	1.00	1.01
uber	0.000	0.998	4	0.000	0.454	4	0.99	2.67
nips	0.016	0.416	7	0.004	0.435	7	1.01	1.17
enron	0.037	0.031	8	0.045	0.030	8	1.16	1.08
flickr4d	0.358	0.014	15	0.148	0.020	12	1.26	0.95
deli4d	0.797	0.009	16	0.596	0.010	15	1.14	0.82

5.3.7 Reordering Overhead

Figure 5.15 shows the overhead of LEXI-ORDER with 3 iterations. We specifically report the ratio of LEXI-ORDER ordering time to HICOO construction time. (That is, we show how much more expensive it might be to reorder compare to constructing the HICOO representation in the first place.) The results lie in the range of 1.5 to 12.0×. Thus, an important area for future work is to reduce this reordering overhead, possibly by merging the reordering process with HICOO construction as well as using thread parallelization for the reordering itself, which was done sequentially in these experiments.

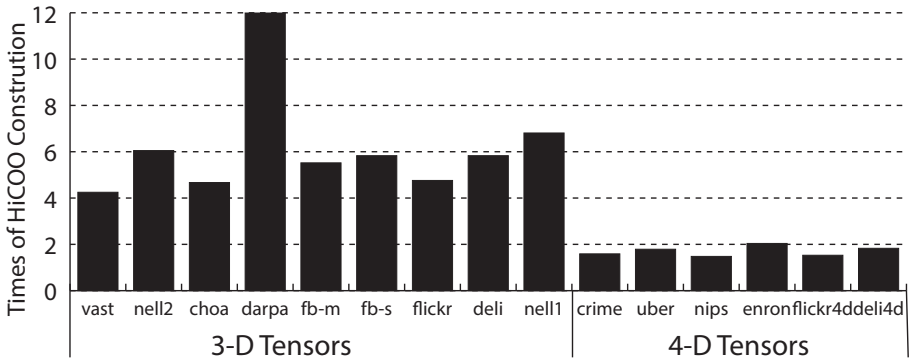


Figure 5.15: The reordering overhead over HICOO construction.

5.4 Related Work

Various reordering methods have been considered for matrix algebra, such as [5, 49, 71, 72, 82, 83, 115, 116, 139–142, 184, 185]. For tensor operations, Shaden et al. [156] model the memory access of the MTTKRP operation, while Kaya et al. [85] model computational load and communication volume of MTTKRP using hypergraphs. These two hypergraphs are constructed differently from our BFS-LIKE-MCS, they use fibers and tasks as vertices respectively. Though they both use hypergraph partitioning methods, Shaden et al. [156] induce a tensor reordering while Kaya et al. [85] generate a tensor partitioning on distributed processors. Since our BFS-LIKE-MCS approach only takes connectivity into account and does not consider memory access pattern, it does not behave well on all formats and almost all tensors. However, our LEXI-ORDER performs comparably with the hypergraph partitioning method used by Shaden et al. [156] and is not mode dependent.

5.5 Summary

Motivated by an interest in further improving a HICOO implementation, we believe the problem of reordering to improve block density for tensor computations is a novel and interesting one. LEXI-ORDER obtains large MTTKRP performance speedup for both sequential and multicore parallel implementations on HICOO and COO formats and marginal speedup on CSF format; while BFS-LIKE-MCS does not improve much MTTKRP perfor-

mance. Users can control a parameter of LEXI-ORDER to reduce the reordering overhead without much performance drop of MTTKRP and also choose the optimal HICOO parameters for the reordered tensors to pursue the highest MTTKRP performance.

And while the improvements are large primarily in the sequential, rather than the parallel, case, the "baseline" is the tensor in its natural order *after* preprocessing by others. Since we do not know the provenance of the tensors or how they were assembled, it is possible there could be even stronger benefits in settings where there is little or no "natural" ordering structure as the tensor input arrives. In any case, we regard the study of this chapter as an initial foray with many opportunities for future work.

CHAPTER 6

INTENSLI: AN INPUT-ADAPTIVE AND IN-PLACE DENSE TTM

In this chapter, we consider the problem of how to improve the single-node performance of a dense *mode-n product* [89], or more simply, a dense TTM which multiplies a dense tensor by a dense matrix (Section 2.2.2). For some tensor decompositions, e.g., Tucker decomposition, TTM is both a fundamental building block and usually the dominant bottleneck.

The conventional way to implement TTM relies on *matricization* of the tensor, which essentially transforms the tensor into an equivalent matrix [89] (Section 2.1.1). TTM then simply becomes a GEMM. This approach is sensible, for two reasons. First, GEMM is a clean and portable abstraction for which many fast implementations already exist [48, 66, 168, 176]. Secondly, the basic theoretical computation and communication complexity of TTM matches GEMM [159]. Thus, TTM has high arithmetic intensity at sufficiently large sizes and so should scale well, just like GEMM.

Unfortunately, we observe that state-of-the-art GEMM-based implementations of TTM, such as the TENSOR TOOLBOX or CTF [162], can perform well *below* what one expects from GEMM alone (§ 6.1). The problem is matricization.

Mathematically, matricization is merely a conceptual (or logical) restructuring of the tensor, as noted in Section 2.1.1. However, this step is almost always implemented as an explicit copy. Copying ensures that the matricized tensor has a non-unit stride in only one dimension, as required by the Basic Linear Algebra Subprograms (BLAS) [24]. By contrast, doing the TTM in-place *without* such copies would require a different interface for GEMM, one which could support non-unit strides in *both* the row and column dimensions [168]; accordingly, performance may be expected to decrease due to decreases in cache line utilization, prefetch bandwidth, and the degree of SIMDization. (Copying can also roughly double the storage; see § 6.1.) Nevertheless, the conventional wisdom is that this copy need

not incur a *performance* overhead. By contrast, we find that explicit matricization can in practice *dominate* the overall running time, accounting for 70% of the total running time or more, even at large sizes[1] (figure 6.2 in § 6.1).

These observations raise a very natural question: can one achieve GEMM-like performance for TTM using only an *in-place* approach?

We answer the preceding question affirmatively. Our main contribution is a novel framework for automatically generating high-performance in-place dense TTM implementations (§ 6.2). This framework generates TTM implementations that use coarse-grained parallelism via OpenMP and any underlying high-performance GEMM. It considers several strategies for decomposing the specific type of TTM that the user requires in order to maximize the use of the fast GEMM. We show that our framework can produce TTM implementations that achieve a high fraction of what a hypothetical pure GEMM can achieve (§ 6.4).

Beyond a code generation strategy, the second contribution of our work is a method, based on empirical model-based tuning, to select a good implementation of the TTM. Our method considers the characteristics of the input to the TTM, such as the size of the tensor in each dimension. Put differently, the code generation framework produces several parameterized implementations; and we use a heuristic model to select the parameters, which need to be tuned for a given input. In this way, the framework's results are *input-adaptive* [105]. Thus, even if a compiler-based loop transformation system can, from a naïve TTM code as input, produce the codes that our framework generates, the input-adaptive aspect of our approach can be considered a distinct contribution.

We have implemented our approach as a system called INTENSLI, which is pronounced as "intensely" and is intended to evoke an *In-place and input-adaptive Tensor Library*.[2]

[1] The argument would be that the copy is, asymptotically, a negligible low-order term in the overall running time. However, that argument does not hold in many common cases, such as the case of the output tensor being is much smaller than the input tensor. This case arises in machine learning and other data analysis applications, and the result is that the cost of a data copy is no longer negligible.

[2] This work has been published [100].

Taken together, we show that our INTENSLI-generated TTM codes can outperform the TTM implementations available in two widely used tools, the TENSOR TOOLBOX [16] and CTF [162], by about 4 times and 13 times, respectively. Our framework can be directly applied to tensor decompositions, such as the well-known Tucker decomposition, whose computation heavily relies on efficient tensor-times-matrix multiply [89].

Table 6.1 summarizes the symbols and notations used in this chapter. For other general symbols, refer to table 2.1.

Table 6.1: List of symbols and notations in Chapter 6.

Symbols	Description
$\mathbf{X}_{\text{SUB}}, \mathbf{Y}_{\text{SUB}}$	Sub-tensors
m, n, k	Matrix sizes
Q	#Floating-point operations (or Flops)
W	#Words
$A, \hat{A}$	Arithmetic intensity
Input parameters	
P	#CPU threads
Z	Last-level cache size in words
INTENSLI parameters	
M_L	A set of loop modes
M_C	A set of component modes
N_C	#Component modes in M_C, $N_C = \|M_C\|$
P_L	Loop parallel degree
P_C	MM parallel degree
θ_{MS}	Minimum storage size threshold of a MM kernel for N_C
θ_{ML}	Maximum storage size threshold of a MM kernel for N_C
θ_P	Storage size threshold of a MM kernel for thread allocation

6.1 Motivating Observations

Currently, most tensor toolkits, including the widely used TENSOR TOOLBOX [16] and CTF libraries [160], implement TTM using a three-step method: (i) matricizing the tensor by *physically* reorganizing it (copying it) into a matrix; (ii) carrying out the matrix-matrix product, using a fast GEMM implementation; and (iii) tensorizing the result, again by physical reorganization. This procedure appears in algorithm 10 and figure 6.1.

To motivate the techniques proposed in § 6.2, we make a few observations about the

Algorithm 10 The baseline mode-n product algorithm (TTM) in Tensor Toolbox [16] and CTF [160].

Input: A dense tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a matrix $\mathbf{U} \in \mathbb{R}^{R \times I_n}$, and an integer n;
Output: A dense tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \times \cdots \times I_N}$;

// **Matricize**: Transform from $\mathcal{X}$ to $\mathbf{X}_{(n)}$.
1: Set sz = size($\mathcal{X}$), order = $[n,\ 1:n-1,\ n+1:N]$;
2: $\widetilde{\mathcal{X}} = \text{permute}(\mathcal{X}, \text{order})$;
3: $\mathbf{X}_{\text{mat}} = \text{reshape}(\widetilde{\mathcal{X}})$, $\mathbf{X}_{\text{mat}} \in \mathbb{R}^{I_n \times \prod_{-n}^{i=1,\ldots,N} I_i}$;
// **Multiply**: $\mathbf{Y}_{(n)} = \mathbf{U}\mathbf{X}_{(n)}$.
4: $\mathbf{Y}_{\text{mat}} = \mathbf{U} * \mathbf{X}_{\text{mat}}$;
// **Tensorize**: Transform from $\mathbf{X}_{(n)}$ to $\mathcal{X}$.
5: Set out_sz = $[R, \text{sz}(1:n-1), \text{sz}(n+1:N)]$;
6: $\widetilde{\mathcal{Y}} = \text{tensor}(\mathbf{Y}_{\text{mat}}, \text{out\_sz})$;
7: $\mathcal{Y} = \text{inversePermute}(\widetilde{\mathcal{Y}}, \text{order})$;
8: **return** $\mathcal{Y}$;

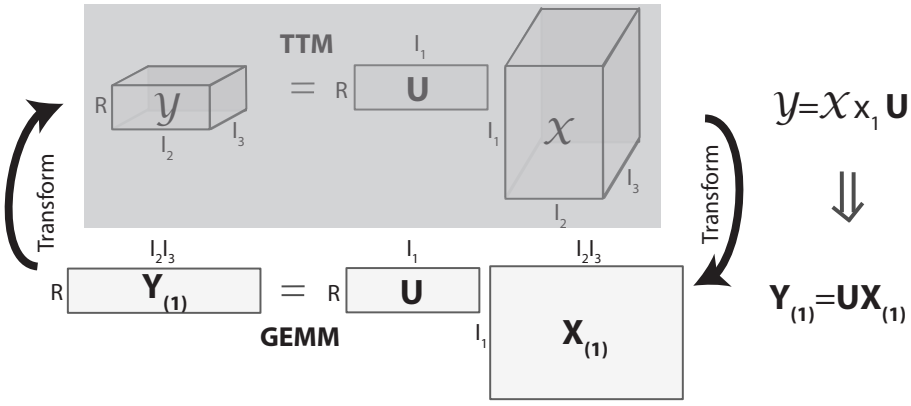
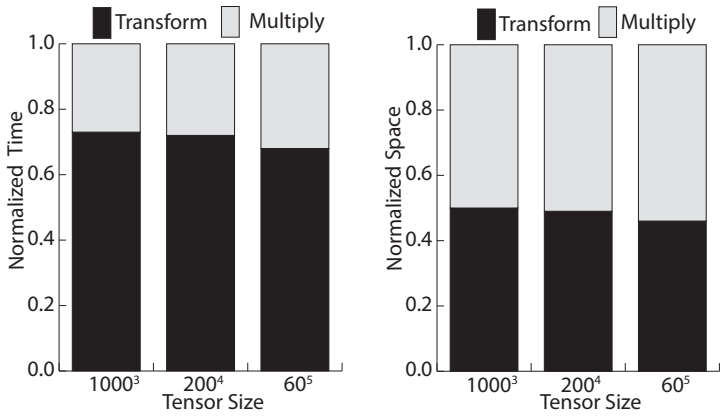


Figure 6.1: Structure of the baseline mode-1 TTM computation.

traditional implementations of TTM (algorithm 10). These observations reveal several interesting problems and immediately suggest possible ways to improve TTM performance.

Observation 1: *In a traditional* TTM *implementation, copy overheads dominate when the output tensor is much smaller than the input tensor.* Consider algorithm 10, which has two transformation steps, one on lines 2–3 and the other on lines 6–7. They can carry a large overhead when implemented as physical reorganizations, i.e., explicit copies.

For instance, consider the profiling results for a mode-2 product on 3rd-order, 4th-order, and 5th-order dense tensors, as shown in figure 6.2. The x-axis represents tensor sizes and the y-axis shows the fraction of time or space of the transformation step (lower dark bars)



(a) Time Profiling (b) Space Profiling

Figure 6.2: Profiling of algorithm 10 for mode-2 product on 3rd, 4th, and 5th-order tensors, where the output tensors are low-rank representations of corresponding input tensors.

compared to the matrix multiply (upper light-grey bars). These profiles were gathered for the TENSOR TOOLBOX implementation.[3] Transformation takes about 70% of the total running time and accounts for 50% of the total storage.

To see fundamentally why copying hurts performance, compare the arithmetic intensity of TTM with copy overheads against GEMM. Consider a system with a two-level memory hierarchy, that is, with a large slow memory connected to a fast memory (e.g., last-level cache) of size Z words. Recall that the *arithmetic intensity* of a computation running on such a system is the ratio of its floating-point operations, or *flops* Q, to the number of words, W, moved between slow and fast memory [76]; the higher a computation's intensity, the closer to peak it will run. For both GEMM and tensor contractions, $W \geq \frac{Q}{8\sqrt{Z}} - Z$ [17]. Thus, an upper-bound on arithmetic intensity, A, is

$$A \equiv \frac{Q}{W} \quad \leq \quad \frac{Q}{\frac{Q}{8\sqrt{Z}} - Z} \approx 8\sqrt{Z}, \tag{6.1}$$

where the latter bound holds if $\frac{Q}{8\sqrt{Z}} \gg Z$, or $Q \gg 8Z^{3/2}$. For a non-Strassen GEMM on $n \times n$ matrices, $Q = 2n^3$, so that equation (6.1) holds when $n^3 \gg 4Z^{3/2}$. Assuming a

[3] TENSOR TOOLBOX calls Intel Math Kernel Library as the high-performance BLAS implementation of GEMM.

cache of size 8 MiB, which is 2^{20} double-precision words, then $4Z^{3/2} = 2^{32}$, which means the matrix dimension n should satisfy $n \gtrsim 1600$.

Now consider a TTM implementation that copies explicitly. Suppose this TTM involves an order-N tensor of size I in each dimension, so that the tensor's size is I^N and the TTM's flops are $\hat{Q} = 2I^{N+1}$. The two transformations steps together require moving a total of $2I^N$ words. Further suppose that this TTM does the *same* number of flops as the preceding GEMM; then, $\hat{Q} = Q$ or $I = n^{\frac{3}{N+1}}$. The intensity $\hat{A}$ of this TTM will be, again assuming $\hat{Q} = Q \gg 8Z^{3/2}$,

$$\hat{A} \lesssim \frac{\hat{Q}}{\frac{\hat{Q}}{8\sqrt{Z}} + \frac{\hat{Q}}{I}} = \frac{8\sqrt{Z}}{1 + \frac{8\sqrt{Z}}{I}} \approx \frac{A}{1 + \frac{A}{I}}. \tag{6.2}$$

Thus, copying might reduce the copy-free intensity, A, by a factor of $1 + A/I$. How large can this penalty be? If $n \gtrsim 1600$ and $N = 3$, then $I \approx 254$ and $1 + A/I \approx 33$. This penalty increases as the order N increases. Thus, one should avoid explicit copying.

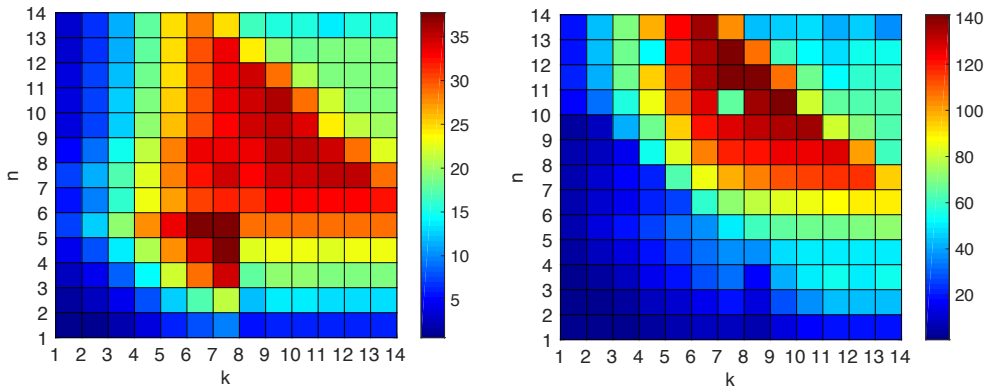
Observation 2: *The* GEMM *sizes that arise in* TTM *may be far below peak.* It is well-known that while GEMM benchmarks emphasize its performance on large and square problem sizes, the problem sizes that arise in practical applications are often rectangular and, therefore, may have a very different performance profile. The best example is LU decomposition, in which one repeatedly multiplies a "tall-skinny" matrix by a "short-fat" matrix.[4] Similarly, one expects rectangular sizes to arise in TTM as well, due to the nature of the decomposition algorithms that invoke it. That is, the GEMM on line 4 of algorithm 10 will multiply operands of varying and rectangular sizes. A particularly common case is one in which $R \ll I_i$, which can reduce intensity relative to the case of $R \approx I_i$.

Based on these expectations, we measured the performance of a highly-tuned GEMM implementation from Intel's Math Kernel Library [48]. Figure 6.3 shows the results. The operation was $\mathbf{C} = \mathbf{B}\mathbf{A}^T$, where $\mathbf{A} \in \mathbb{R}^{n \times k}$, $\mathbf{B} \in \mathbb{R}^{m \times k}$. The value of m is fixed to be

[4] That is, block outer products.

$m = 16$, which corresponds to a "small" value of R. Each square is the performance at a particular value of k (x-axis, shown as $\log_2 k$) and n (y-axis, shown as $\log_2 n$), color-coded by its performance in GFLOP/s. Figure 6.3(a) shows the case of a single thread, and (b) for 4 threads. (The system is a single-socket quad-core platform, described in § 6.4.)

According to figure 6.3, performance can vary by roughly a factor of 6, depending on the operand sizes. When k or n becomes very large, performance can actually decrease. The maximum performance in figure 6.3 is about 38 GFLOP/s for 1 thread, and 140 GFLOP/s for 4 threads. This particular GEMM can actually achieve up to 50 GFLOP/s and 185 GFLOP/s for 1 and 4 threads, respectively, at large square sizes.



(a) m=16, using 1 thread (b) m=16, using 4 threads

Figure 6.3: Performance of dense, double-precision general matrix multiply from the Intel Math Kernel Library for $\mathbf{C} = \mathbf{B}\mathbf{A}^T$, where $\mathbf{A}$ is $n \times k$, $\mathbf{B}$ is $m \times k$. The value of m is fixed at 16, while k (x-axis) and n (y-axis) vary. Note that the x- and y-axis labels show $\log_2 k$ and $\log_2 n$, respectively. Each square is color-coded by performance in GFLOP/s.

Observation 3: *Analogous to* BLAS *operations, different ways of organizing* TTM *have inherently different levels of locality.* There are several ways to organize a TTM. Some of these formulations lead to scalar operations, which have relatively poor locality compared to Level-2 (matrix-vector) or to Level-3 (matrix-matrix) type operations. As with linear algebra operations, one should prefer Level-3 organizations over Level-2 or Level-1.

We summarize these different formulations in table 6.2, for the specific example of computing a mode-1 TTM on a third-order tensor. There are two major categories. The first is based on *full reorganization* (algorithm 10), which transforms the entire input ten-

Table 6.2: A third-order tensor's different representation forms of mode-1 TTM. The input tensor is $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, the input matrix is $\mathbf{U} \in \mathbb{R}^{I_1 \times R}$, and the output tensor is $\mathcal{Y} \in \mathbb{R}^{R \times I_2 \times I_3}$.

Mode-1 Product Representation Forms		BLAS Level	Transformation
Full reorganization	*Tensor representation* $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}$	—	—
	Matrix representation $\mathbf{Y}_{(1)} = \mathbf{U}\mathbf{X}_{(1)}$	L3	Yes
Sub-tensor extraction	*Scalar representation* $y_{ri_2i_3} = \sum_{i_1=1}^{I_1} x_{i_1 i_2 i_3} u_{ri_1}$ $Loops: i_{2(3)} = 1, \cdots, I_{2(3)}, r = 1, \cdots, R$	Slow	No
	Fiber representation $\mathbf{y}(r, :, i_3) = \mathbf{X}(:, :, i_3)\mathbf{u}(r, :)$, $Loops: i_3 = 1, \cdots, I_3, r = 1, \cdots, R$	L2	No
	Slice representation $\mathbf{Y}(:, :, i_3) = \mathbf{U}\mathbf{X}(:, :, i_3), Loop: i_3 = 1, \cdots, I_3$	L3	No

sor into another form before carrying out the product (§ 2.1.1). This category includes the matricization approach. The second category is based on *sub-tensor extraction*. The formulations in this category iterate over sub-tensors of different shapes, such as scalars, fibers, or slices (§ 2.1.1). By analogy to the BLAS, each of the formulations corresponds to the BLAS level shown in the table (column 3). The full reorganizations also imply at least a logical transformation of the input tensor, whereas sub-tensor extraction methods do not (column 4). Further explanation of different representations will be given in § 6.2.

6.2 In-Place and Input-Adaptive TTM

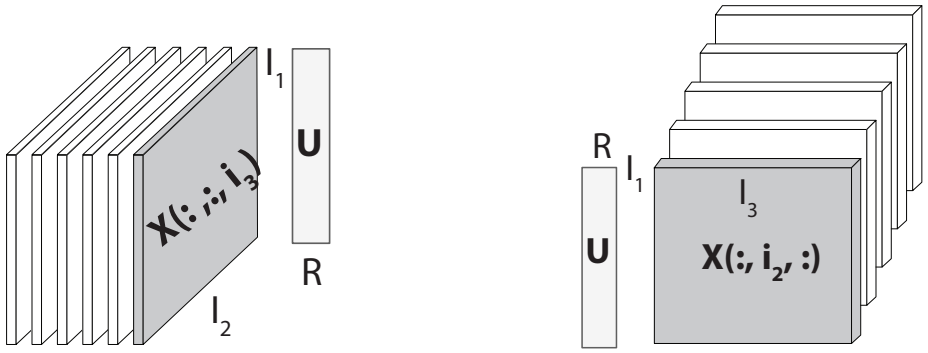
This section describes the INTENSLI framework, which automatically generates an in-place, input-adpative TTM (INTTM). By in-place, we mean avoiding the need for physical reorganization of the tensor or its sub-tensors when computing the TTM; by input-adaptive, we mean choosing a near-optimal strategy for a given set of inputs, which for a mode-n product include the value of n and the size of the input tensor (both order and length of each dimension).

6.2.1 A Third-Order Tensor Example

To build some intuition for our scheme, we introduce the basic terms on a third-order tensor. But the strategies and conclusions from this section is applicable to arbitrary-order tensors.

From § 2.1.2, there are several dense representations for a tensor operation: tensor, matrix, slice, fiber, and scalar representations from two major categories. *Full reorganization* transforms the entire input tensor into another form before carrying out the product; while *sub-tensor extraction* iterates over sub-tensors of different shapes. We use TTM as an example to show the different formulations of them in table 6.2. Consider a third-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ and a matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, with which we wish to compute the mode-1 product (or TTM), $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}$. The matrix representation of TTM first matricizes tensor $\mathcal{X}$ to $\mathbf{X}_{(1)}$, and does a matrix-matrix multiplication to compute $\mathcal{Y}_{(1)}$, then converts it back to tensor $\mathcal{Y}$. The two transformations are non-trivial because of their large amount of memory access. The slice representation fixes one mode index of $\mathcal{Y}$, say i_3, with an loop iterating all its values; the fiber representation fixes the indices of two modes, say j and i_3, with two loops iterating all their values; and the scalar representation fixes the indices of all modes inside three nested loops. Matrix representation requires twice transformations, while slice, fiber, and scalar representations can be performed *in-place* so long as we pre-allocate $\mathcal{Y}$. By analogy to the BLAS, the formulations corresponds to different BLAS levels. Typically, operations in a higher BLAS level can potentially achieve higher performance. The full reorganizations also imply at least a logical transformation of the input tensor, whereas sub-tensor extraction methods do not. Therefore, the slice representation is the most performance and storage efficient in table 6.2.

Moreover, for the slice representation we are also free to choose which mode(s) to fix and which to iterate over. Alternatively, we may compute $\mathcal{Y}$ using another slice representation by computing $\mathbf{Y}(:, i_2, :) = \mathbf{U}\mathbf{X}(:, i_2, :)$ for all $i_2 = 1, \cdots, I_2$ too. We use the term *stride* refers to the distance in physical memory between two consecutive elements along one mode. The last-mode dominated layout in § 2.1.2, analogous to row-major layout for



(a) $\mathbf{Y}(:,:,i_3) = \mathbf{U}\mathbf{X}(:,:,i_3)$, non-contiguous. (b) $\mathbf{Y}(:,i_2,:) = \mathbf{U}\mathbf{X}(:,i_2,:)$, contiguous.

Figure 6.4: Two examples of the slice representation.

matrices, has the unit stride for the last mode which is called *leading mode*. For instance, suppose our third-order tensor $\mathcal{X}$ is stored in the last-mode dominated layout. Then, each element x_{ijk} is stored at position $i_1 \cdot (I_2 I_3) + i_2 \cdot (I_3) + i_3$. The leading mode is the third mode. We take two examples of the slice representation, $\mathbf{Y}(:,:,i_3) = \mathbf{U}\mathbf{X}(:,:,i_3)$ and $\mathbf{Y}(:,i_2,:) = \mathbf{U}\mathbf{X}(:,i_2,:)$ in figure 6.4. The row stride of slice $\mathbf{X}(:,:,i_3)$ is $I_2 I_3$ and the column stride is I_3, while the row stride of slice $\mathbf{X}(:,i_2,:)$ is $I_2 I_3$ and the column stride is 1. If both row and column strides are non-unit, we call the slice (or matrix) is stored in a *general stride layout*.

The traditional BLAS interface for matrix-matrix multiplication (GEMM) operations *require* unit stride in at least one dimension, so users cannot operate on operands stored in a general stride layout. The BLAS-like Library Instantiation Software (BLIS) framework develops interfaces to support generalized matrix storage [168]. However, GEMM on strided matrices is generally expected to be slower than GEMM on regular matrices, since strided memory access can easily under-utilize cache lines compared to contiguous memory access. Therefore, a reasonable implementation heuristic is to avoid non-unit stride computation. For tensor computations, that also means *we should prefer to build or reference sub-tensors starting with the leading mode*. Thus, we prefer the slice representation $\mathbf{Y}(:,i_2,:) = \mathbf{U}\mathbf{X}(:,i_2,:)$ to $\mathbf{Y}(:,:,i_3) = \mathbf{U}\mathbf{X}(:,:,i_3)$.

We can also reshape a tensor into a matrix by partitioning its indices into two disjoint

subsets, in different ways. For example, the tensor $\mathcal{X}(:,:,:) \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ can be reshaped into a matrix $\widetilde{\mathbf{X}}(:,:) \in \mathbb{R}^{(I_1 \times I_2) \times I_3}$, that is, a matrix with $I_1 I_2$ rows and I_3 columns. This particular reshaping is purely logical: it does not require physically reorganizing the underlying storage before passing it either to a BLAS-style GEMM that requires a unit-stride matrix or a BLIS-style GEMM that supports a general stride matrix (see § 6.2.2). Another way to reshape $\mathcal{X}$ is to generate a matrix $\hat{\mathbf{X}}(:,:) \in \mathbb{R}^{I_2 \times (I_1 \times I_3)}$. As it happens, this reshape operation is impossible without physically reorganizing the data (see § 6.2.2). *Thus, we need to be careful when choosing a new dimension order of a tensor, to avoid physical reorganization.* These two facts are the key ideas behind our overall strategy for performing TTM in-place.

6.2.2 Algorithmic Strategy

We state two lemmas: the first suggests how to build matrices given an arbitrary number of modes of the input tensor, and the second establishes the correctness of computing a matrix-matrix multiplication on sub-tensors.

Lemma 6.2.1. *Given an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and a matrix $\mathbf{U} \in \mathbb{R}^{R \times I_n}$, the mode-$n$ product can be performed without physical reorganization on up to $\max\{n-1, N-n\}$ contiguous dimensions.*

Proof. From section § 6.2.1, to avoid physical reorganization, the dimensions of sub-tensors should be a sub-sequence of the input tensor's. Combining contiguous dimensions is the only way to build a high performance MM kernel with sub-tensors (matrices).

We first prove that two contiguous modes (except mode-n) can be reshaped to one dimension without physical reorganization. Let m and $m+1$ be two contiguous modes. Let $\widetilde{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_{new} \cdots \times I_N}$, where $I_{new} = I_m I_{m+1}$, be the reshaped tensor that comes from combining these two modes into a new dimension. Because the order of the two contiguous modes is unchanged, elements of tensor $\widetilde{\mathcal{X}}$ have exactly the same physical arrangement as tensor $\mathcal{X}$. Thus, we can logically form tensor $\widetilde{\mathcal{X}}$ from $\mathcal{X}$ without data movement. However,

for two non-contiguous modes, e.g., mode-m and $(m+2)$, we cannot form a new reshaped tensor without a permutation, which demands physically reorganizing $\mathcal{X}$.

When reshaping contiguous modes, the resulting sub-tensors $\mathbf{X}_{\text{SUB}} \in \mathbb{R}^{I_n \times I_{new}}$ are actually matrices. For a mode-n product, the most contiguous modes are obtained by either the leftmost modes of mode-n, $\{1, \cdots, n-1\}$ or the rightmost ones $\{n+1, \cdots, N\}$. Thus, up to $\max\{n-1, N-n\}$ contiguous modes can be reshaped into a "long" single dimension in the formed matrices, without physical data reorganization. □

Lemma 6.2.2 applies lemma 6.2.1 to the computation of the mode-n product.

Lemma 6.2.2. *Consider the mode-n product, involving an order-N tensor. It can be computed by a sequence of matrix multiplies involving sub-tensors formed either from the leftmost contiguous modes, $\{1, \ldots, m_1\}$, $1 \leq m_1 \leq n-1$; or the rightmost contiguous modes, $\{m_2, \ldots, N\}$, $n+1 \leq m_2 \leq N$.*

Proof. Recall the scalar definition of the mode-n product, $\mathcal{Y} = \mathcal{X} \times_n \mathbf{U}$, which is

$$y_{i_1 \cdots i_{n-1} r \cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 \cdots i_N} u_{r i_n}.$$

Without loss of generality, consider the rightmost contiguous modes $\{m_2, \cdots, N\}$, and suppose they are combined into a single dimension. Let $I_p \equiv I_{m_2} \cdots I_N$ be the total size of this new dimension. Then, we may extract two new (logical) sub-tensors (matrices): $\mathbf{X}_{\text{SUB}}$, which is taken from $\mathcal{X}$ and has size $I_n \times I_p$, and $\mathbf{Y}_{\text{SUB}}$, which is taken from $\mathcal{Y}$ and has size $R \times I_p$. The GEMM operation, $\mathbf{Y}_{\text{SUB}} = \mathbf{U}\mathbf{X}_{\text{SUB}}$, is in scalar form, $(Y_{\text{sub}})_{r i_p} = \sum_{i_n=1}^{I_n} u_{r i_n} (X_{\text{sub}})_{i_n, i_p}$. The index $i_p = i_{m_2} \times I_{m_2+1} \cdots I_N + \cdots + i_N$, corresponds to the offset $(i_{m_2}, \cdots, i_N)$ in the tensors $\mathcal{X}$ and $\mathcal{Y}$; the remaining modes, $\{i_1, \cdots, i_{n-1}, i_{n+1}, \cdots, i_{m_2-1}\}$, are fixed indices during this GEMM. Thus, iterating over all possible values of $\{i_1, \cdots, i_{n-1}, i_{n+1}, \cdots, i_{m_2-1}\}$ and performing the corresponding matrix multiplies yields the same result as the mode-n product. □

Lemmas 6.2.1 and 6.2.2 imply an alternative representation of a mode-n product, relative to those listed in table 6.2. This algorithmic strategy may be summarized by the INTTM procedure shown in algorithm 11 and a sixth-order example in figure 6.5.

Algorithm 11 In-place Tensor-Times-Matrix Multiply (INTTM) algorithm to compute a mode-n product. "inplace-mat" means in-place building a sub-tensor (matrix) from initial full tensor, using modes for its row and column respectively.

Input: A dense tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{R \times I_n}$, and an integer n;
Output: A dense tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \times \cdots \times I_N}$;

▷ Nested loops, using P_L threads

1: **parfor** $i_l = 1$ to I_l, all $i_l \in M_L$ **do**
2: **if** M_C are on the left of i_n **then**
3: $\mathbf{X}_{\text{SUB}}$ = inplace-mat ($\mathcal{X}$, M_C, i_n);
4: $\mathbf{Y}_{\text{SUB}}$ = inplace-mat ($\mathcal{Y}$, M_C, r);
▷ Matrix-matrix multiplication, using P_C threads
5: $\mathbf{Y}_{\text{SUB}} = \mathbf{X}_{\text{SUB}}\mathbf{U}'$, $\mathbf{U}'$ is the transpose of $\mathbf{U}$.
6: **else**
7: $\mathbf{X}_{\text{SUB}}$ = inplace-mat ($\mathcal{X}$, i_n, M_C);
8: $\mathbf{Y}_{\text{SUB}}$ = inplace-mat ($\mathcal{Y}$, r, M_C);
▷ Matrix-matrix multiplication, using P_C threads
9: $\mathbf{Y}_{\text{SUB}} = \mathbf{U}\mathbf{X}_{\text{SUB}}$
10: **end parfor**
11: **return** $\mathcal{Y}$;

To instantiate this algorithm for a tensor with given dimensions, we must generate a set of nested loops (line 1) and use a proper kernel for the inner matrix-matrix multiplication (lines 5 and 9).

A number of parameters arise as a result of this algorithm:

- *Loop modes,* M_L. Modes utilized in nested loops. They are not involved in the inner-most matrix multiplies. For instance, in figure 6.5, the loop modes are i_1, i_2.
- *Component modes,* M_C. Modes participate in matrix multiply. M_L and M_C constitute the set of all tensor modes, except mode-n. From lemma 6.2.1, only contiguous modes are considered as component modes. In figure 6.5, component modes are i_4, i_5, i_6.

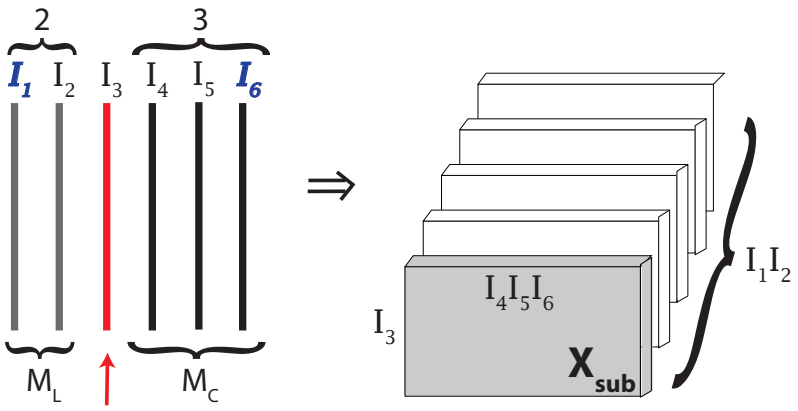


Figure 6.5: Structure of new mode-3 INTTM computation for a sixth-order tensor using forward strategy.

- *Loop parallel degree,* P_L. The number of threads utilized by nested loops, i.e., loop modes, to exploit parallelization at a coarse-grained level.

- *MM parallel degree,* P_C. The number of threads employed by the inner-most matrix multiply; fine-grained parallelism.

In algorithm 11, we avoid physical reorganization of the $\mathcal{X}$ and $\mathcal{Y}$ tensors, and build a MM kernel accordingly using the chosen component modes (M_C). If M_C is assigned to the left modes of i_n (called *backward strategy*), $\mathbf{X}_{\text{SUB}}$ and $\mathbf{Y}_{\text{SUB}}$ are in-place constructed with the size of $\prod_{i_c \in M_C} I_c \times I_n$ and $\prod_{i_c \in M_C} I_c \times R$ respectively. Mode-n is taken as the columns of $\mathbf{X}_{\text{SUB}}$ and $\mathbf{Y}_{\text{SUB}}$, thus we use $\mathbf{Y}_{\text{SUB}} = \mathbf{X}_{\text{SUB}}\mathbf{U}'$ as the MM kernel. Otherwise, M_C is assigned to the modes to the right of i_n (called *forward strategy*), mode-n is taken as the rows of $\mathbf{X}_{\text{SUB}}$ and $\mathbf{Y}_{\text{SUB}}$. Thus we use $\mathbf{Y}_{\text{SUB}} = \mathbf{U}\mathbf{X}_{\text{SUB}}$ as the MM kernel. Note that sub-tensors (matrices) are not explicitly built, but implicitly referred to logically sub-tensors. Multi-threaded parallelism is additionally employed on both nested loops and the MM kernel, exposing two additional parameters. The parameter configuration will be described in the next section.

By avoiding an explicit data copy, the intensity $\widetilde{A}$ of INTTM algorithm will be,

$$\widetilde{A} \lesssim \frac{\hat{Q}}{\frac{\hat{Q}}{8\sqrt{Z}}} = 8\sqrt{Z} \approx A. \tag{6.3}$$

Our in-place INTTM algorithm improves the arithmetic intensity of tensor-times-matrix multiply, by eliminating the factor $1 + \frac{A}{I}$. The arithmetic intensity of INTTM is close to GEMM, so it has the potential to achieve comparable performance to GEMM. Furthermore, utilizing in-place operations decreases storage space by approximately 50%.

The INTTM algorithm also presents new challenges. *Challenge 1:* INTTM does not have a natural static representation. As shown in algorithm 11, loop modes M_L and component modes M_C vary with the input tensor and mode-n. Because its performance might depend on the input, INTTM algorithm is a natural candidate for code-generation and auto-tuning. *Challenge 2:* INTTM algorithm may operate on inputs in a variety of shapes, as opposed to only square matrices. For instance, it would be common in INTTM in the case of a third-order tensor for two of the matrix dimensions to be relatively small compared to the third. Additionally, there might be large strides.

Despite these challenges, we still have opportunities for optimization. To this end, we build an input-adaptive framework that generates code given general input parameters. We also embed optimizations to determine the four parameter values M_L, M_C, P_L, and P_C in this framework, to generate an optimal tensor-times-matrix multiply code.

6.3 An Input Adaptive Framework

Our input-adaptive framework is shown in figure 6.6, illustrating the procedure of generating INTTM for a given tensor. There are three stages: input, parameter estimation, and loop generation, which generates the INTTM code. Input parameters include data layout, input tensor, leading mode, MM Benchmark, and the maximum supported number of threads. The parameter estimator predicts the optimal values of intermediate parameters, including

loop order, M_L, M_C, P_L, and P_C. These intermediate parameters guide the generation of INTTM code. Within INTTM, either the BLIS or MKL libraries will be called according to the parameter configuration.

We first illustrate parameter estimation, then explain the code generation process in the following sections.

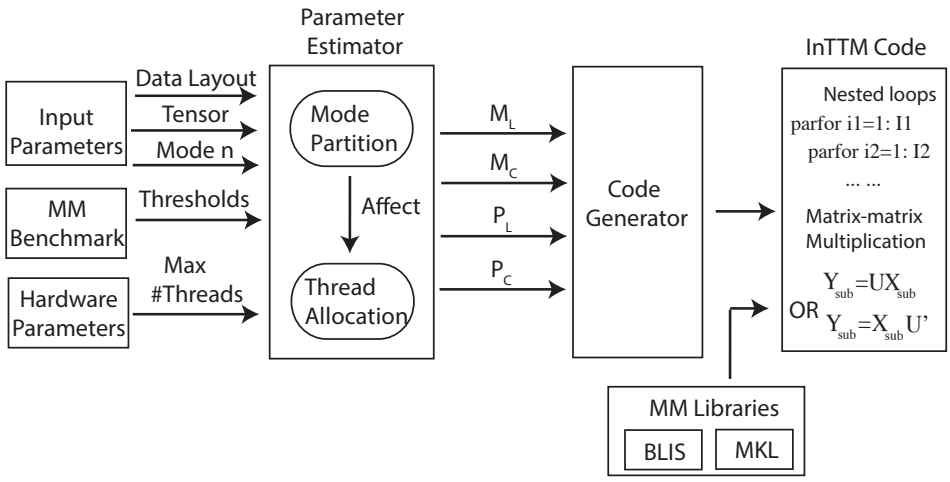


Figure 6.6: Input adaptive framework (INTENSLI).

6.3.1 Parameter Estimation

The two main aspects of parameter estimation are mode partitioning and thread allocation. Mode partitioning is the most important, and its decision influences thread allocation, so we state it first.

Mode partitioning. Apart from mode-n, all tensor modes are partitioned into two sets M_L and M_C, to generate nested loops and sub-tensors for inner matrix multiply. From figure 6.3, matrices in different sizes achieve very different performance numbers. In mode partitioning, we primarily decide M_C to maximize MM kernel's performance, and the rest modes are assigned to M_L.

Lemma 6.2.1 implies that there are two ways to choose contiguous modes, namely, from the modes either to the right or left of mode-n, forward and backward strategies, respectively. Forward strategy is shown using a sixth-order example in figure 6.5. In the

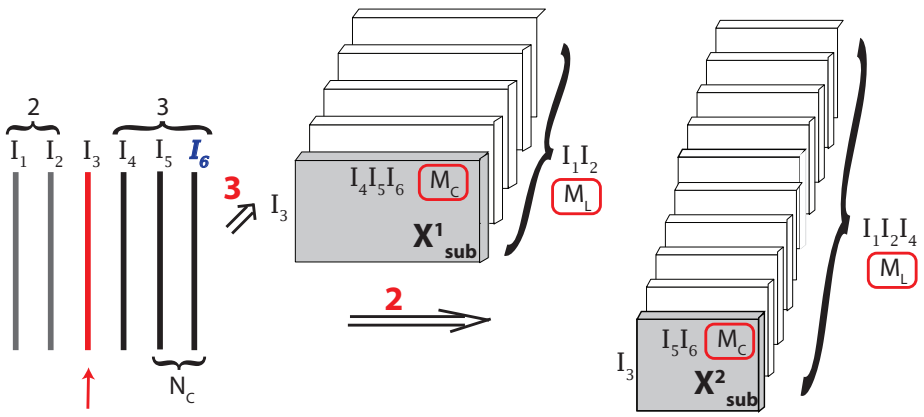


Figure 6.7: Mode-3 INTTM for a sixth-order tensor using forward strategy under different N_C settings.

forward strategy, the mode set is $M_C = \{m_2, \cdots, N\}$, where $n+1 \leq m_2 \leq N$; whereas in the backward strategy, the mode set $M_C = \{1, \cdots, m_1\}$, where $1 \leq m_1 \leq n-1$. If a tensor is stored in row-major pattern, the forward strategy generates a MM kernel with unit stride access in one dimension, while the backward strategy employs a general stride MM kernel. This means using forward strategy MM kernel can call the BLAS, but the backward strategy would need general stride support, as provided by BLIS. Different stride sizes and MM kernel implementations affect INTTM performance. As one would expect and experiments confirm, it is usually not possible for BLIS operating on large general strides to achieve performance comparable to MKL when one dimension has unit stride. In the experiments below, we assume row-major layout, in which case we adopt the forward strategy. (One would use the backward strategy if the default layout were assumed to be column-major.)

After determining the strategy, the parameter m_1 (or m_2) in lemma 6.2.1 also needs to be determined. We introduce another parameter, N_C, which specifies the number of component modes in M_C. Once N_C is decided, so is m_1 (or m_2). Two examples of a sixth-order INTTM with different N_Cs using forward strategy is plotted in figure 6.7. For instance, take a sixth-order tensor, $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_6}$ and a matrix $\mathbf{U} \in \mathbb{R}^{R \times I_3}$, and suppose we wish to compute mode-3 product using the forward strategy. When $N_C = 3$, $m_2 = 4$,

$M_C = \{i_4, i_5, i_6\}$, $\mathbf{X}_{\text{SUB}} \in \mathbb{R}^{I_3 \times I_4 I_5 I_6}$. When $N_C = 2$, $m_2 = 5$, $M_C = \{i_5, i_6\}$, $\mathbf{X}_{\text{SUB}} \in \mathbb{R}^{I_3 \times I_5 I_6}$. Thus, different values of N_C imply different sub-tensors and MM kernel sizes. As shown in figure 6.3, one would therefore expect performance to vary with different values of k and n, when m is fixed to a small value. Based on this observation, in our scheme we build a MM benchmark and use it to generate two thresholds, θ_{MS} and θ_{ML}, which are used to determine the optimal N_C and then M_C. To determine the thresholds, there are two steps. The first step is to fix k while varying n, since m is generally fixed when the tensors arise in some low-rank decomposition algorithm; and the second step is varying k.

Figure 6.8 shows MM performance on different values of n, when m and k are fixed. This figure shows a clear trend: after performance reaches a top-most value, as n increases, MM performance rebounds. We see a similar trend with other values of m and k. Matrices with unbalanced dimensions do not achieve high performance, partially because it is more difficult to apply a multilevel blocking strategy, as in GotoBLAS [65, 66]. Put differently, when one dimension is relatively small, there may not actually be enough computation to hide data transfer time.

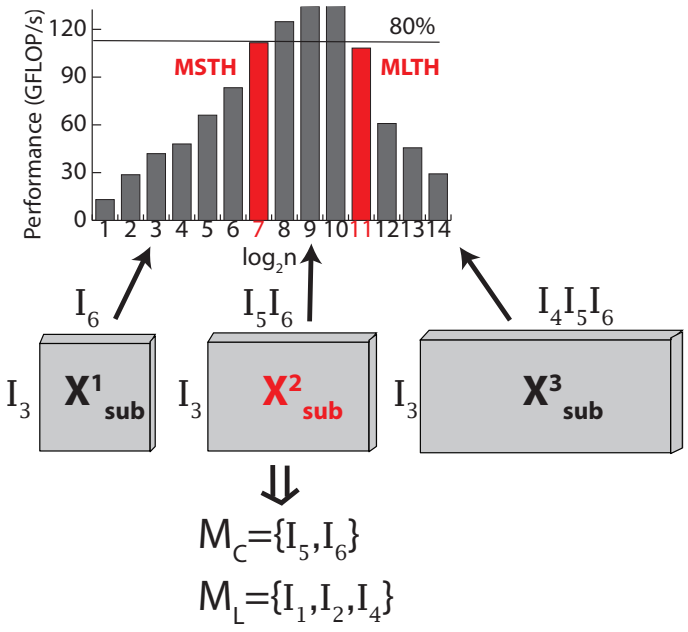


Figure 6.8: Performance variation of MM on different sizes of n, when $m = 16$, $k = 512$, using 4 threads. MM kernels with different N_Cs are also shown.

Figure 6.8 has a typical shape, which is some function $f(n)$ having a peak, $f_{\max}$. Consider a horizontal line, $\kappa f_{\max}$, for some fraction $\kappa = 0.8$, chosen empirically. The thresholds θ_{MS} and θ_{ML} correspond to the two red bars closest to but below the horizontal line. We set θ_{MS} (θ_{ML}) as the storage size of the three matrices of the MM kernel with chosen n values. The second step is to calculate the average of each threshold over different values of k. We wish to find the MM kernel with the three matrices' storage size between the computed values of θ_{MS} and θ_{ML}, which is relatively more likely to produce high GEMM performance. The size bounds of the MM kernel limit the N_C parameter in our INTTM algorithm. Under different N_Cs (a.k.a. M_Cs), the size of n varies a lot, the best of which is $N_C = 2$, $M_C = \{i_5, i_6\}$. From our experiments on a particular machine (with a Core i7 processor in our experiment), θ_{MS} is evaluated to 1.04 MB and θ_{ML} is evaluated to 7.04 MB.

The parameter N_C is first initialized to 1, and its MM kernel's storage size is calculated. If it is smaller than θ_{MS}, we increment N_C and re-evaluate the storage size until we find the maximum MM kernel size between θ_{MS} and θ_{ML}. Now, we use N_C to partition modes to M_L and M_C.

From figure 6.6, the data layout also affects the partitioning process. Recall that the assumed data layout affects the choice of a forward (row-major) versus a backward (column-major) strategy. So, if $N_C = p$, M_C is $\{N - p + 1, \cdots, N\}$ (row-major) or $\{1, \cdots, p\}$ (column-major). This choice in turn means that the component modes should be chosen from the leading dimension, to guarantee unit stride in one dimension. The data layout also decides the order of M_L modes, where loop order is an increasing sequence of dimensions for row-major pattern.

Thread allocation. After determining M_C, we decide P_L and P_C according to the MM kernel size. From our tests, if the matrix is small, parallelizing the nested loops is more efficient than parallelizing the matrix multiply. For large matrices, the situation is the opposite. We use a threshold θ_P for thread allocation, which is also shown as the storage size of the MM kernel. The difference is that θ_P is determined from INTTM experiments,

not from MM benchmark. If the size of MM kernel is smaller than θ_P, we allocate more threads to nested loops; otherwise, more threads are allocated to MM kernel. The value of θ_P is set to 800 KB in our tests. From our experiments, the highest performance is always achieved when using maximum threads on either nested loops or the MM kernel, so we only consider these two situations.

6.3.2 Code Generation

The code generation process consists of two pieces: generating nested loops and generating wrappers for the matrix multiply kernel. For each mode in set M_L, we build a for loop for it according to the mode order established by M_L. P_L is the number of threads allocated in nested loops. Code is generated in C++, using OpenMP with the collapse directive.

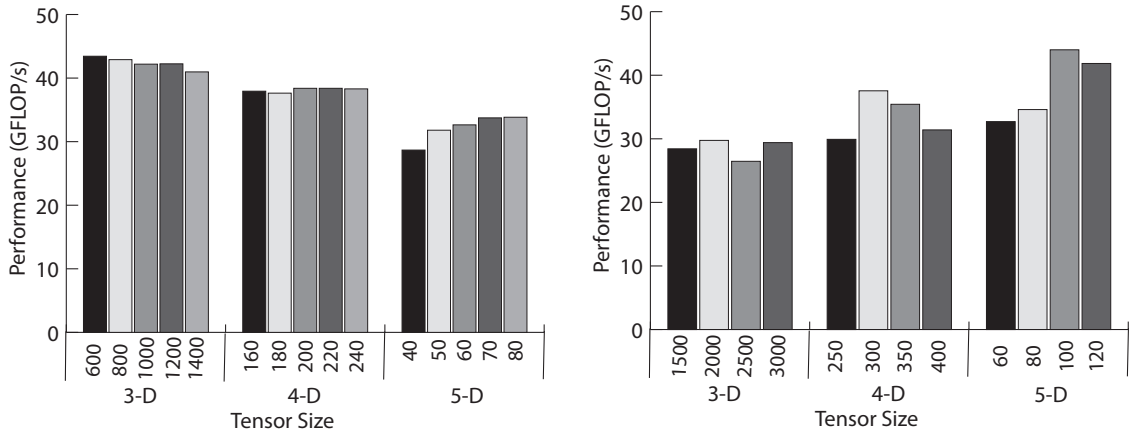
For the matrix multiply kernel, we build in-place sub-tensors $\mathbf{X}_{\text{SUB}}$ and $\mathbf{Y}_{\text{SUB}}$ using modes in M_C. According to the row and column strides, we choose between BLIS [168] or Intel MKL libraries [48]. Thus, a complete INTTM code is generated according to the determined parameters.

6.4 Experiments and Analyses

Our experimental evaluation focuses on three aspects of INTENSLI: (a) assessing the absolute performance (GFLOP/s) of its generated implementations; (b) comparing the framework against widely used alternatives; and (c) verifying that its parameter tuning heuristics are effective. This evaluation is based on experiments carried out on the two platforms shown in table 6.3, one based on a Core i7-4770K processor and the other on a Xeon E7-4820. Our experiments employ 8 and 32 threads on the two platforms respectively, considering hyper-threading. The system based on the Xeon E7-4820 has a relatively large memory (512 GiB), allowing us to test a much larger range of (dense) tensor sizes than has been common in prior single-node studies. Note that all computations are performed in double-precision.

Table 6.3: Experimental Platforms Configuration

Parameters	Intel Core i7-4770K	Intel Xeon E7-4820
Microarchitecture	Haswell	Westmere
Frequency	3.5 GHz	2.0 GHz
#Physical cores	4	16
Hyper-threading	On	On
Peak GFLOP/s	224	128
Last-level cache	8 GiB	18 GiB
Memory size	32 GiB	512 GiB
Memory bandwidth	25.6 GB/s	34.2 GB/s
#Memory channels	2	4
Compiler	icc 15.0.2	icc 15.0.0



(a) Intel Core i7-4770K (b) Intel Xeon E7-4820

Figure 6.9: Performance of INTENSLI-generated INTTM algorithm for mode-2 product on 3rd, 4th, and 5th-order tensors. Each bar represents the performance of a specific tensor size.

6.4.1 Basic benchmark.

We first check that the INTENSLI-generated INTTM delivers consistent performance at a variety of tensor dimensions and sizes. The results for a mode-2 product, as an example, appear in figure 6.9. We tested $\mathcal{X} \times_2 \mathbf{U}$ where $\mathcal{X}$ is an order-N tensor of size I in each dimension (total tensor size is I^N) and $\mathbf{U}$ is a $16 \times I$ matrix to agree with the low-rank property of tensor decomposition. We test $N \in \{3, 4, 5\}$ at various values of I, shown along the x-axis, chosen so that the largest I still permits I^N to fit into main memory; the y-axis shows performance in GFLOP/s. On the Core i7, our INTTM achieves over 40 GFLOP/s on 3rd-order tensors, with performance tending to steadily decrease or remain flat with increasing size and order. At higher orders, dimension size decreases in order to fit into memory, which reduces the inner GEMM performance as we would expect from the observations of § 6.1. By contrast, performance trends on the Xeon E7 platform differ from those on the Core i7. In particular, 3rd-order tensors show the worst performance, compared to higher-order tensors. This stems in part from worse Intel MKL performance on the Xeon E7—it achieves only 51 GFLOP/s on a square GEMM with operands of size 1000×1000, compared to 154 GFLOP/s on the Core i7. However, it also happens that, on the Xeon E7, the multithreading within MKL does not appear to benefit GEMM performance much. Our ability to achieve higher performance with higher orders on that platform comes mainly from our use of coarse-grained outer-loop parallelization.

On both platforms, INTENSLI-generated INTTM does not deliver performance that compares well with the peak performance shown in table 6.3. The main reason is, as noted in § 6.1 and figure 6.3, GEMM performance differs from peak for rectangular shapes. Such shapes can arise in INTENSLI-generated TTM code, since it considers a variety of ways to aggregate and partition loops.

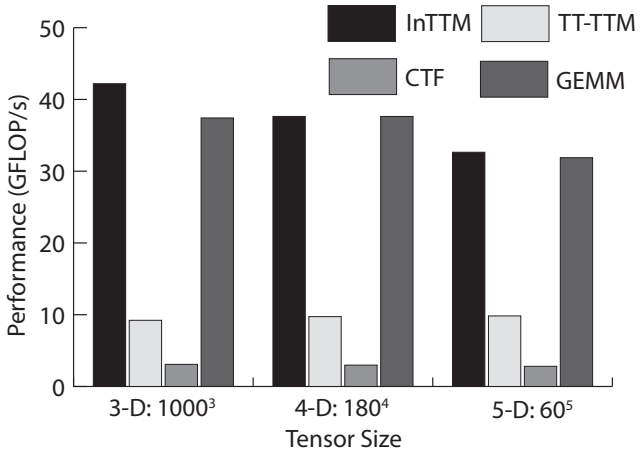


Figure 6.10: Performance comparison among INTENSLI-generated INTTM, TENSOR TOOLBOX (TT-TTM), Cyclops Tensor Framework (CTF), and GEMM on 3rd, 4th, and 5th-order tensors of mode-2 product.

6.4.2 Comparison to other tools.

We compare INTENSLI-generated INTTM code against the equivalent implementations in the TENSOR TOOLBOX and CTF. Note that the TENSOR TOOLBOX and CTF implementations use algorithm 10. TENSOR TOOLBOX is designed to be MATLAB-callable, but under-the-hood uses multithreading and can link against a highly-tuned BLAS. CTF is implemented in C++, with OpenMP parallelization and BLAS calls where possible. In addition, it is useful to also compare against GEMM using a matricized tensor (line 4 of algorithm 10) but *ignoring* any reorganization time and other overhead. This measurement of just GEMM provides an estimate of the performance we might expect to achieve. TENSOR TOOLBOX, CTF, GEMM, and our INTTM are all linked to MKL library for GEMM calls.

The results appear in figure 6.10. Because TENSOR TOOLBOX and CTF have larger memory requirements than the INTTM, the tensor sizes selected for figure 6.10 are smaller than for figure 6.9. In figure 6.10, the leftmost bar is INTENSLI-generated INTTM, which achieves the highest performance among the four. The performance of TENSOR TOOLBOX and CTF is relatively low, at about 10 GFLOP/s and 3 GFLOP/s, respectively. Our INTTM gets about $4\times$ and $13\times$ speedups compared to TENSOR TOOLBOX and CTF. The

main reason is that TENSOR TOOLBOX and CTF incur overheads from explicit copies. The rightmost bar is GEMM-only performance. Our INTTM matches it, and can even outperform it since the INTENSLI framework considers a larger space of implementation options than what is possible through algorithm 11.

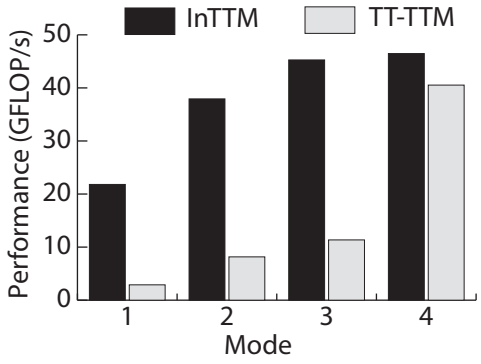


Figure 6.11: Performance behavior of INTENSLI-generated INTTM against TENSOR TOOLBOX (TT-TTM) for different mode products on a $160 \times 160 \times 160 \times 160$ 4th-order tensor.

We also compare against just TENSOR TOOLBOX for varying mode-n computations in figure 6.11, on a 4th-order tensor.[5] X-axis shows the modes corresponding to our INTTM algorithm. The TENSOR TOOLBOX performance varies significantly among different modes, ranging from 3 GFLOP/s to 40 GFLOP/s. By contrast, the INTENSLI-TTM reduces this performance variability with changing mode. This result shows the benefit of specializing the partitioning and iteration-ordering strategy, as INTENSLI does automatically.

6.4.3 Parameter selection.

The last major aspect of the INTENSLI framework is selecting good parameters. Recall that during the mode partitioning process, a GEMM benchmark is used to determine two thresholds, θ_{MS} and θ_{ML}. In figure 6.12, we compare the results of using INTENSLI's heuristics to choose these parameters against an exhaustive search, in the case of a mode-1 product

[5]The TENSOR TOOLBOX uses a column-major ordering, whereas INTENSLI uses a row-major ordering. As such, for a order-N tensor, we compare TENSOR TOOLBOX's mode-n product against our mode-$(N - n + 1)$ product, i.e., their mode-1 against our mode-4 product, their mode-2 against our mode-3 product, and so on. In this way, the storage pattern effect is eliminated.

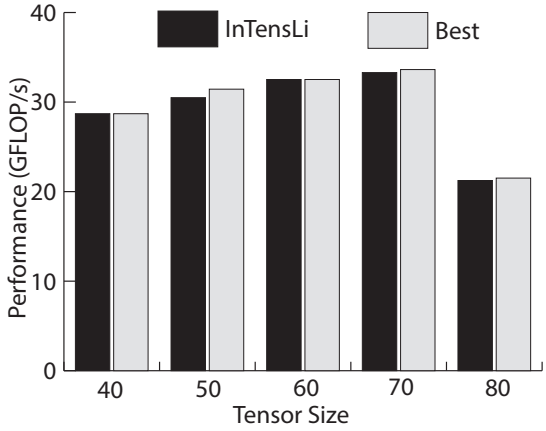


Figure 6.12: Comparison between the performance with predicted configuration and the actual highest performance on 5th-order tensors of mode-1 product.

on a 5th-order tensor. The black bars show the performance of the configuration chosen automatically by INTENSLI, and the gray bars show the performance of the configuration chosen by exhaustive search. For this particular input, there are 16 possible configurations, among which INTENSLI's heuristics select just 1. As figure 6.12 shows, INTENSLI makes near-optimal choices.

From these experiments, INTENSLI can easily be integrated in tensor decompositions, such as Tucker decomposition for low-order tensors and hierarchical Tucker for high-dimensional tensors. Such applications can benefit from the performance and space-efficiency of INTENSLI-generated code.

6.5 Related Work

One of the most widely-used tensor implementations is the TENSOR TOOLBOX. Its optimized TTM kernel has been the baseline comparison for our experiments. This implementation utilizes an algorithm (METTM) [91] that alleviates the intermediate memory-blowup problem. By carefully choosing which modes to compute with finer granularity, the intermediate data remains within working memory. TENSOR TOOLBOX suffers from excessive data copy according to our experiments, which motivates our in-place approach.

Cyclops Tensor Framework (CTF) [162] provides another baseline implementation. CTF is a recent HPC implementation with two levels of parallelism (OpenMP and MPI) which focuses on communication reduction for symmetric tensor contractions that arise frequently in quantum chemistry. TTM is a specific instance of tensor contraction. CTF distributes tensors via a slice-representation. Another implementation that specializes on contraction is the Tensor Contraction Engine (TCE) [10], a mature implementation that focuses on synthesizing code and dynamically maintains load balance on distributed systems. TCE also builds a model to choose optimal data layout, while we choose from different matrix shapes. INTENSLI-generated INTTM can serve as a single-node implementation for a distributed version.

As discussed in table 6.2, there are many different ways to think about and represent the same tensor operation, and there has been a recent flurry of work on rethinking tensor representations to make for more efficient, scalable decompositions.

The Matricized Tensor Times Khatri-Rao Product (MTTKRP) is an essential step of CANDECOMP/PARAFAC (CP) Decomposition, and differs from general TTM in that the matrix is the result of the Khatri-Rao product of two matrices. N.Ravindran, et al. created an in-place tensor-matrix product for MTTKRP [146], but their implementation operates on the "slice" representation of the tensor. Our work takes advantage of a more general subtensor representation, and in particular its opportunities for performance tuning.

Baskaran et al. [21] implement a data structure approach to improving the performance of tensor decompositions, proposing a sparse storage format that can be organized along modes in such a way that data reuse is increased for the Tucker Decomposition. In contrast, our approach avoids the overhead of maintaining a separate data structure and can use native, optimized multiply operations. Di Napoli et al. [56] establish precise conditions to determine if and when using GEMM for a tensor contraction. Our approach additionally supplies auto-tuning method to further improve the performance.

6.6 Summary

The key finding of this INTENSLI work is that a mode-n product, or TTM operation, can be performed efficiently in-place and tuned automatically to the order and dimensions of the input tensor. Our INTENSLI framework can serve as a template for other primitives and tensor products, a few of which appear in § 6.5. Although we focused on improving single-node performance, including exploiting shared memory parallelism and reducing TTM bandwidth requirements, this building block can be used as a "drop-in" replacement for the intra-node compute component of distributed memory implementations.

CHAPTER 7

SPARSE TTM AND TUCKER DECOMPOSITION ON CPU-GPU PLATFORMS

This chapter considers the problem of optimizing the tensor-times-dense matrix (TTM) operation (Section 2.2.2) in the case that the input tensor is *sparse*. Regarding this chapter's scope, we consider thread parallelism and memory locality for single-node GPU platforms.

In principle, a sparse TTM is similar to a sparse matrix-times-dense matrix. Conventional SPTTM implementations, such as those in TENSOR TOOLBOX [16, 89] and Cyclops Tensor Framework (CTF) [162], first transform a sparse tensor into an equivalent sparse matrix and then assume an optimized sparse matrix-times-dense matrix to invoke. This approach is a reasonable way to utilize an existing and highly efficient state-of-the-art sparse matrix-times-dense matrix subroutine. However, just as in the case of dense TTM from Chapter 6, this conversion step incurs non-trivial cost in time and space. Furthermore, the generated matrix could be extremely large in one dimension, the explicit indexing of which—for a many-way tensor—might quickly exceed the range of a 64-bit unsigned integer. Therefore, we are motivated to avoid any such conversion, carrying out the sparse TTM "natively" on a given input tensor.

The main use-case for TTM is the Tucker decomposition, which invokes two types of sparse TTMs: a general sparse TTM (SPTTM) and semi-sparse TTM (SSPTTM) (see § 7.4). SPTTM, as we know, is a general sparse tensor times a dense matrix; while SSPTTM is a semi-sparse tensor times a dense matrix. A semi-sparse tensor is a dense-structured tensor that at least one mode is dense. For example, a tensor with its first mode dense means the fibers of this mode are either empty or fully dense. For a semi-sparse tensor, as observed by Baskaran et al. [21], its dense substructure can be explored for better performance compared to the general sparse case. Here, we consider performance optimizations for both SPTTM and SSPTTM, primarily with for GPU-based platforms.

Our proposed techniques make the following contributions.

- First, we design an in-place sequential SPTTM algorithm to avoid tensor-matrix data transformation, which is based on a structured sparse tensor format and a certain special property. Additionally, an auxiliary array is used to avoid memory write conflict for the later-on parallel SPTTM algorithms. Our sequential SPTTM is $3 - 120\times$ faster than the SPTTM from TENSOR TOOLBOX library [16]. (§ 7.1 and § 7.2)

- Secondly, we parallelize SPTTM on single-node GPU systems. We propose several optimizing approaches for SPTTM on NVIDIA GPUs: employing fine thread granularity, arranging coalesced memory access, rank blocking, and using local (fast) memory ("shared memory" on GPUs). GPU-SPTTM obtains $6 - 19\times$ speedup on NVIDIA K40c and $23 - 67\times$ speedup on NVIDIA P100 over CPU-SPTTM respectively for real-world tensors. Our GPU-SPTTM is $3.9\times$ faster than the state-of-the-art GPU implementation. (§ 7.3)

- Thirdly, we implement SSPTTM on GPUs accordingly by better exploring the dense sub-structures. Our SSPTTM implementations outperform SPTTMs which handles the input semi-sparse tensor in a general way by $4.5\times$. (§ 7.4)

- Lastly, by applying SPTTM and SSPTTM to the Tucker decomposition, it outperforms CPU Tucker decomposition by $3.2\times$. (§ 7.5)

Parts of this work have been published in our previous paper [103]. The approach using fine thread granularity is the same with non-shared memory GPU SpTTM, while our SpTTM approach using shared memory in this chapter is an optimized version thereof. However, the work of this chapter extends that paper in four ways. 1) This work further optimizes SPTTM on NVIDIA GPUs by employing five optimization approaches and providing a more in-depth analysis of their performance. 2) SSPTTM optimizations on these platforms are also explored, which further speed up TTM operations in Tucker decomposition. 3) We built a parallel Tucker decomposition for GPUs by applying the optimized

SPTTM and SSPTTM. 4) Afterwards, these algorithms are tested on an extended sparse tensor set, including both third- and fourth-order real-world tensors. One consequence is that our new GPU optimizations achieve much better speedup than in our prior publication.

Table 7.1 summarizes the symbols and notation of this chapter, with other general symbols listed in table 2.1.

Table 7.1: List of symbols and notations in Chapter 7.

Symbols	Description
Input parameters	
N_{TB}	Maximum #Threads per block
S_{SM}	Maximum shared memory size in words
N_{TR}	Maximum #Threads per block for a matrix row
Algorithm parameters	
n_{fibs}	#Mode-n fibers of $\mathcal{X}$
f_{ptr}	The beginnings of $\mathcal{X}$'s mode-n fibers, sized n_{fibs}
f_{len}	The average length of $\mathcal{X}$'s mode-n fibers, $f_{\mathrm{len}} = \frac{M}{n_{\mathrm{fibs}}}$
$D_{\mathcal{Y}}$	Dense mode set of a semi-sparse tensor
$S_{\mathcal{Y}}$	Sparse mode set of a semi-sparse tensor
n_{chunks}	#$D_{\mathcal{Y}}$-chunks of $\mathcal{Y}$
c_{ptr}	The beginnings of $\mathcal{Y}$'s $D_{\mathcal{Y}}$-chunks, sized n_{chunks}

7.1 Semi-sparse Tensor Format (sCOO)

We study "semi-sparse" tensors which are sparse tensors with at least one mode dense, since they are intermediate results of the tensor decompositions with dense factor matrices, thus an efficient format is critical to a tensor decomposition's performance (details in § 2.3). Different from § 3.2.3, we use a simple semi-COO (sCOO) format, an auxiliary format of COO format, which only stores the indices of sparse modes and all the non-zero values in a particular order for dense modes. The reused indices between a sparse tensor in COO and a semi-sparse tensor in sCOO are only saved once. The idea of distinguishing dense and sparse modes was first proposed by Baskaran et al. [21] as "Mode-generic sparse storage format". Figure 7.1 shows a sCOO representation of an example semi-sparse tensor with dense mode 3.

Assume integer indices and floating-point values have the same size of 32 bits, sCOO format saves 50% space of COO format for this toy example tensor. sCOO format saves at least $\frac{N_D}{N+1}$ storage space for an Nth-order semi-sparse tensor with N_D dense modes. The storage saving comes from that sCOO does not store the indices of dense modes, and the indices of sparse modes are compressed in sCOO than in COO.

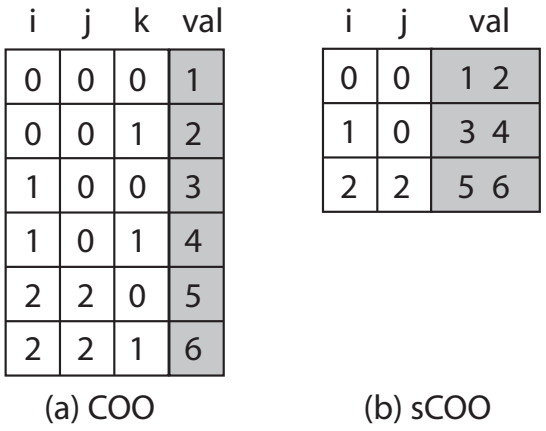


Figure 7.1: COO and sCOO formats of a semi-sparse $3 \times 3 \times 2$ tensor, with dense mode 3.

7.2 SpTTM on CPUs

Based on COO and sCOO formats, we implement sequential SpTTM by directly operating on non-zero entries without explicit transformation between a tensor and a matrix.

Given a sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, we know the resulting tensor $\mathcal{Y}$ is a semi-sparse tensor from the above property (§ 3.2.2). The intuitive algorithm for SpTTM without explicit transformation is to loop all non-zeros of $\mathcal{X}$ by multiplying each with its corresponding row of $\mathbf{U}$. Then all rows having the same index pair $(i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N)$ are sum-reduced to get a fiber of $\mathcal{Y}$ ($\mathbf{f}_n^Y$).

This algorithm has two issues: First, in the sum-reduction stage, there is an implicit index comparing operation even if $\mathcal{X}$ is pre-sorted. The complexity of the index comparison is high especially for higher-order tensors. An extra comparison operation for an additional mode increases the SpTTM complexity by M^X (the number of non-zeros of $\mathcal{X}$), which is

non-trivial especially for low-rank tensor decompositions with a small R. Second, the sum-reduction stage is hard to parallelize and may lead to severe memory contention.

To solve these problems, we design our sequential SPTTM algorithm (Algorithm 12) to avoid expensive index comparison and with inherent good parallelism. Each mode-n fiber of $\mathcal{Y}$ ($\mathbf{f}_n^Y$) is a sized-R dense vector, we record n_{fibs} as the number of $\mathbf{f}_n^Y$. Then the number of non-zeros of $\mathcal{Y}$: $M^Y = n_{\text{fibs}} \times R$. We use an extra array f_{ptr} to identify the beginning locations of every mode-n fiber of $\mathcal{X}$ ($\mathbf{f}_n^X$), then iterate all n_{fibs} fibers of $\mathcal{Y}$. Comparing to iterating $\mathcal{X}$ in the intuitive algorithm, we avoid the expensive index comparison in the sum-reduction stage.

Our SPTTM has two stages, preprocessing and computing. The preprocessing stage includes three steps: sorting $\mathcal{X}$, calculating f_{ptr}, and pre-allocating semi-sparse tensor $\mathcal{Y}$. $\mathcal{X}$ is first sorted according to the product mode n, then f_{ptr} is allocated and calculated to identify the beginning locations of n_{fibs} mode-n fibers of $\mathcal{X}$ ($\mathbf{f}_n^X$). From SpTTM's property (§ 3.2.2), the semi-sparse tensor's index modes remain unchanged, so the number of $\mathbf{f}_n^Y$ is also n_{fibs}. Based on the pre-sorted $\mathcal{X}$, we pre-allocate the exact space for $\mathcal{Y}$ and only for its non-zero entries, because $\mathcal{X}$'s $(N-1)$ indices can be reused by $\mathcal{Y}$. During the computation stage of Algorithm 12, each $\mathbf{f}_n^Y = \text{val}^Y(i,:)$ locates the corresponding $\mathbf{f}_n^X = \text{val}^X(f_{\text{ptr}}(i), \ldots, f_{\text{ptr}}(i+1)-1)$. Then $\mathbf{f}_n^Y$ is the sum of rows $\mathbf{u}(k,:)$ scaled by each non-zero of fiber $\mathbf{f}_n^X$.

Algorithm 12 CPU sequential SPTTM algorithm (SEQ-SPTTM).

Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, and an integer n;
Output: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \cdots \times I_N}$;

1: n_{fibs}: the number of mode-n fibers of $\mathcal{Y}$
2: f_{ptr}: the beginnings of each $\mathcal{X}$ mode-n fiber, size n_{fibs}.
3: **for** $i = 0, \ldots, n_{\text{fibs}}$ **do**
4: **for** $j = f_{\text{ptr}}(i), \ldots, f_{\text{ptr}}(i+1) - 1$ **do**
5: $k = ind_n^X(j)$
6: **for** $r = 0, \ldots, R-1$ **do**
7: $val^Y(i,r) += val^X(j) \times u(k,r)$
8: **Return** $\mathcal{Y}$;

The number of floating-point operations (flops) of sequential SPTTM (Algorithm 12) is

$$flops = 2 \cdot M^X \cdot R, \tag{7.1}$$

where M^X is the number of non-zeros of $\mathcal{X}$. Algorithm 12 eliminates the index comparison. For a third-order sparse tensor, our SPTTM only uses n_{fibs} extra space for f_{ptr}. Since $n_{\text{fibs}} \leq M^X$, the extra space is much smaller than the matriced tensor of $\mathcal{X}$ $(3 \cdot M^X)$ in the traditional algorithms where tensor transformation is needed.

We parallelize SPTTM on the multicore CPU architecture using OpenMP. Since our sequential SPTTM iterates all independent fibers of $\mathcal{Y}$, we can easily parallelize this loop by assigning CPU threads, i.e., parallelize the outermost for loop. Each thread computes a size-R fiber $\mathbf{f}_n^Y$ independently and shares dense matrix $\mathbf{U}$. Because our SPTTM algorithm limits the sum-reduction dependency inside a thread, parallelized SPTTM naturally avoids locks and utilizes CPU caches well for writing $\mathcal{Y}$. We use CPU-SPTTM to represent parallel CPU SPTTM implementation in the following contents.

7.3 SPTTM on GPUs

To fully explain our optimizations on GPUs, we first start by describing a naïve implementation, then propose four more parallelization and optimization approaches for SPTTM by incrementally considering GPU architecture characteristics. The five implementations are: naïve implementation, employing fine thread granularity, arranging coalesced memory access, rank blocking, and using fast shared memory.

In this work, we assume the sparse tensor $\mathcal{X}$ and the dense matrix $\mathbf{U}$ both reside in GPU memory. We assume all the division operations below are fully divisible.

7.3.1 Naïve implementation

As shown in Algorithm 13, we assign each CUDA thread to one mode-n fiber of $\mathcal{Y}$ ($\mathbf{f}_n^X$) and a fiber of $\mathcal{X}$ ($\mathbf{f}_n^X$). Each thread performs multiplication on every non-zero of fiber $\mathbf{f}_n^X$

with its corresponding rows of $\mathbf{u}(k,:)$. When launching the kernel, we set $\mathrm{dimGrid} = N_{\mathrm{TB}}$ and $\mathrm{dimBlock} = n_{\mathrm{fibs}}/N_{\mathrm{TB}}$, where N_{TB} is the number of threads per block. We tune the value of N_{TB} for the best performance.

Algorithm 13 Naïve SpTTM algorithm on GPU (GPU-SPTTM-Naïve).

Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, an integer n, the beginnings of n_{fibs} $\mathcal{X}$ mode-n fiber f_{ptr}, and GPU thread hierarchy $\mathrm{dimGrid} = N_{\mathrm{TB}}$ and $\mathrm{dimBlock} = n_{\mathrm{fibs}}/N_{\mathrm{TB}}$;

Output: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \cdots \times I_N}$;

1: $\mathrm{tidx} = \mathrm{threadIdx.x}$;
2: $i = \mathrm{blockIdx.x} \times \mathrm{blockDim.x} + \mathrm{tidx}$; ▷ i: global index of a mode-n $\mathcal{Y}$ fiber.
▷ j: global index of the non-zeros of mode-n $\mathcal{X}$ fiber.
3: **for** $j = f_{\mathrm{ptr}}(i), \ldots, f_{\mathrm{ptr}}(i+1) - 1$ **do**
4: $k = \mathrm{inds}_n^X(j)$
5: **for** $r = 0, \ldots, R-1$ **do**
6: $\mathrm{val}^Y(i,r) += \mathrm{val}^X(j) \times u(k,r)$
7: **Return** $\mathcal{Y}$;

Some inefficient spots are observed:

- Uncoalesced and redundant memory transfers. The memory transfers for matrix $\mathbf{U}$ are in a stride-R pattern, and the data transferred are much larger than the size of $\mathbf{U}$ because of irregular memory access.

- Coarse-grained task granularity. Algorithm 13 assigns a fiber per CUDA thread to do f_{len} computations with sized-R rows of $\mathbf{U}$, where f_{len} is the average length of the fibers. For lightweight GPU threads, it might be better to allocate fine-grained tasks for each thread.

- Not utilizing fast memory. Algorithm 13 accesses data only from global memory without well utilized fast scratch-pad memory (shared memory or L1 cache) for writable data, since NVIDIA Kepler and later GPUs cannot use L1 cache for writable data automatically.

To deal with the inefficiency, we propose optimizations described below.

7.3.2 Fine thread granularity

Instead of one-dimensional thread blocks, we assign two-dimensional thread blocks, so both rows and columns of $\mathbf{U}$ are parallelized in Algorithm 14. However, if the matrix row size R is a little large, say 128, the number of threads in x-dimension can be only up to 8, which leads to imbalanced parallelism for x and y dimensions. To prevent this issue, we set a threshold N_{TR} to limit the number of threads allocated to each matrix column, and compute a segment of a column in one iteration.

When launching Algorithm 14, we set $\mathrm{dimBlock} = (N_{\mathrm{TB}}/N_{\mathrm{TR}}, N_{\mathrm{TR}})$ and $\mathrm{dimGrid} = n_{\mathrm{fibs}}/N_{\mathrm{TB}}$.

Algorithm 14 SpTTM algorithm with fine thread granularity on GPU (GPU-SpTTM-FG).

Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, an integer n, the beginnings of n_{fibs} $\mathcal{X}$ mode-n fiber f_{ptr}, and GPU thread hierarchy $\mathrm{dimGrid} = n_{\mathrm{fibs}}/N_{\mathrm{TB}}$ and $\mathrm{dimBlock} = (N_{\mathrm{TB}}/N_{\mathrm{TR}}, N_{\mathrm{TR}})$;

Output: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \cdots \times I_N}$;

1: $n_{\mathrm{loops}}^r = \frac{R}{\mathrm{blockDim}.y}$ ▷ #Iterations for large R.
2: $\mathrm{tidx} = \mathrm{threadIdx.x}$;
3: $\mathrm{tidy} = \mathrm{threadIdx.y}$;
4: $i = \mathrm{blockIdx.x} \times \mathrm{blockDim.x} + \mathrm{tidx}$; ▷ i: global index of a mode-n $\mathcal{Y}$ fiber.
▷ j: global index of the nonzeros of mode-n $\mathcal{X}$ fiber.
5: **for** $j = f_{\mathrm{ptr}}(i), \ldots, f_{\mathrm{ptr}}(i+1) - 1$ **do**
6: $\quad k = \mathrm{inds}_n^X(j)$
7: $\quad$ **for** $l_r = 0, \ldots, n_{\mathrm{loops}}^r$ **do**
8: $\quad\quad r = \mathrm{tidy} + l_r \times \mathrm{blockDim.y}$
9: $\quad\quad \mathrm{val}^Y(i, r) += \mathrm{val}^X(j) \times u(k, r)$
10: **Return** $\mathcal{Y}$;

7.3.3 Coalesced memory access

From Algorithm 14, we observe a memory access pattern where the memory access of tensor indices and values are contiguous and coalesced but that of matrix $\mathbf{U}$ is not. In Algorithm 14, threads in a warp, $(0,0), (1,0), \ldots, (31,0)$ (x-dimension dominates), fetch elements from random addresses because of different k indices, which leads to an uncoalesced memory access. To solve this problem, we swap thread dimensions x and y, thus

x-dimension points to ranks (columns of $\mathbf{U}$) and y-dimension operates on fibers of $\mathcal{X}$ and $\mathcal{Y}$. The algorithm named GPU-SPTTM-MC is shown in Algorithm 14.

When launching this algorithm, we set $\mathrm{dimBlock} = (N_{\mathrm{TR}}, N_{\mathrm{TB}}/N_{\mathrm{TR}})$ and $\mathrm{dimGrid} = n_{\mathrm{fibs}}/N_{\mathrm{TB}}$. Therefore, one warp fetches coalesced global memory addresses of tensor indices, values, and matrix elements.

7.3.4 Rank Blocking

The above algorithms consider the spatial data locality within a fiber, but not the temporal locality between fibers. In Algorithm 14, different fibers may have the same k index, which means accessing the same row of $\mathbf{U}$. Rank blocking strategy is proposed to increase the reuse of row data. Since the computation in-between matrix columns are independent, we exchange the loop order to ensure the loop over ranks (matrix columns) goes before the loop over fiber elements. This strategy enlarges the chance of short rows staying in caches.

Its algorithm (named GPU-SPTTM-RB) can be obtained by swapping tidx and tidy usage and the two loops in Algorithm 14. However, for each batch of short size-N_{TR} matrix rows, the index ks and values of the tensors need to be reloaded from global memory. The benefit of rank blocking depends on the non-zero distribution of the input tensor, it is not easy to determine whether rank blocking is beneficial compared with the above memory coalesced algorithm (GPU-SPTTM-MC).

7.3.5 Using Shared Memory

Obviously, the output val^Y is reused R times for each fiber of $\mathcal{X}$. As we mentioned, the writable val^Y of $\mathcal{Y}$ cannot be cached in the L1 cache for Kepler and later NVIDIA GPU architectures, but can only reside in the slower L2 cache. Therefore, we store the output in shared memory first and then write them back to global memory all at once. In this way, we reduce global memory transfers. Its algorithm is shown in Algorithm 15. When launching this algorithm, we set $\mathrm{dimBlock} = (N_{\mathrm{TR}}, N_{\mathrm{TB}}/N_{\mathrm{TR}})$, $\mathrm{dimGrid} = n_{\mathrm{fibs}}/N_{\mathrm{TB}}$

and set shared memory size $S_{\text{SM}} = N_{\text{TB}}$ words because the size of y_{shr} is $\text{dimBlock.y} \times \text{dimBlock.x}$. Since $N_{\text{TB}} \leq 1024$ for the GPUs we used, the needed shared memory space is smaller than 8KB. That means only the configuration of 48KB L1 cache/16KB shared memory is used for Algorithm 15.

Algorithm 15 SpTTM algorithm using GPU shared memory (GPU-SpTTM-SM).

Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, an integer n, the beginnings of each $\mathcal{X}$ mode-n fiber f_{ptr}, sized n_{fibs}, and GPU thread hierarchy $\text{dimGrid} = N_{\text{TB}}$ and $\text{dimBlock} = (N_{\text{TR}}, N_{\text{TB}}/N_{\text{TR}})$;

Output: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \cdots \times I_N}$;

1: $\mathbf{y}_{\text{shr}}$: shared memory space for $\mathcal{Y}$.
2: $n^r_{\text{loops}} = \frac{R}{\text{blockDim.x}}$
3: $\text{tidx} = \text{threadIdx.x};$
4: $\text{tidy} = \text{threadIdx.y};$
5: $i = \text{blockIdx.x} \times \text{blockDim.y} + \text{tidy};$ ▷ i: global index of a $\mathcal{Y}$ mode-n fiber.
6: **for** $l_r = 0, \ldots, n^r_{\text{loops}}$ **do**
7: $\quad r = \text{tidx} + l_r \times \text{blockDim.x}$
8: $\quad y_{\text{shr}}(\text{tidy}, \text{tidx}) = 0;$
9: $\quad \_\_\text{sync}();$
▷ j: global index of the nonzeros of $\mathcal{X}$ mode-n fiber.
10: $\quad$ **for** $j = f_{\text{ptr}}(i), \ldots, f_{\text{ptr}}(i+1) - 1$ **do**
11: $\quad\quad k = \text{inds}^X_n(j)$
12: $\quad\quad y_{\text{shr}}(\text{tidy}, \text{tidx}) += \text{val}^X(j) \times u(k, r)$
13: $\quad \_\_\text{sync}();$
14: $\quad \text{val}^Y(i, r) = y_{\text{shr}}(\text{tidy}, \text{tidx});$
15: $\quad \_\_\text{sync}();$
16: **Return** $\mathcal{Y}$;

7.4 Semi-Sparse Tensor Times Matrix

SSPTTM is defined as the TTM product of a semi-sparse tensor and a dense matrix, the former being the result of an SPTTM. Although it is possible to convert the semi-sparse tensor back to fully sparse so that SSPTTM can be performed with the above SPTTM algorithm, we propose a tailored SPTTM algorithm optimized specially for semi-sparse tensors.

Given a semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with $D_{\mathcal{Y}} \subset \{1, 2, \cdots, N\}$ being a set of dense modes of $\mathcal{Y}$ and a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, requiring $n \notin D_{\mathcal{Y}}$, the SSPTTM

algorithm also computes $\mathcal{Z} = \mathcal{Y} \times_n \mathbf{U}$. We use $S_\mathcal{Y}$ to represent the sparse modes of $\mathcal{Y}$, $S_\mathcal{Y} = \{1, 2, \cdots, N\} - D_\mathcal{Y}$.

Analogous to SPTTM, our SSPTTM also has two stages, preprocessing and computing. To begin with, we sort values and indices of $\mathcal{Y}$ in a specific mode order s.t. $S_\mathcal{Y} \prec n \prec D_\mathcal{Y}$. For example, if mode $1, 3, 5$ is sparse, mode $2, 4$ is dense, $1 \prec 5 \prec 3 \prec 2 \prec 4$ would be a valid sorting order for an SSPTTM in mode 3. The purpose of this sorting is to gather elements sharing identical sparse mode indices together in a dense "chunk". In the example above, a chunk of $\mathcal{Y}$ is a dense array of size $I_2 \times I_4$, which is a generalization of fibers in SPTTM algorithms. Based on the sparse tensor property (§ 3.2.2), $\mathcal{Z}$ will have the shape of $I_3 \times I_2 \times I_4$ for all chunks by converting mode 3 to a dense mode and enlarging its chunks.

However, different from SPTTM, SSPTTM in Tucker decomposition do not need the expensive sorting step of preprocessing stage. In the Tucker decomposition, an $(N-1)$ TTM-chain (a sequence of TTMs, refer to § 7.5) can be ordered the same with the sorting order of the input sparse tensor $\mathcal{X}$, and an SPTTM or SSPTTM keeps the same order for their sparse modes. Then, only one sorting per TTM-chain for a sparse tensor $\mathcal{X}$ is needed for the first SPTTM. The input semi-sparse tensors for the following SSPTTMs remain sorted. For an Nth-order Tucker decomposition, there are N different sorting orders throughout the entire algorithm, which can be reused among iterations. We cache the N preprocessed tensors in CPU memory and transfer them to GPU memory when necessary to speed up the preprocessing process and save the precious GPU memory.

The CPU parallel algorithm of SSPTTM is Algorithm 16, where the outmost loop-i is parallelized to launch multiple OpenMP threads. For SSPTTM GPU implementations, we map a CUDA thread block into a chunk, because the access pattern of the values inside a $\mathcal{Z}$ chunk only depend on the continuously stored $\mathcal{Y}$ chunks from the previous sorting stage, so inter-chunk calculations can be safely parallelized without data race. In a Tucker decomposition application, where a typical $R = 16$, the number of threads per block can vary from 16 to 65536. The first TTM in the Tucker decomposition is a SPTTM, SSPTTM

handles CUDA block sizes in a typical range of $256 - 65536$, large enough to make full use of the Streaming Multiprocessors. We adjust the block size according to CUDA's restriction of 1024 threads per block, by calculating chunks in batch. The amount of values inside a block is small enough to fit into L1 / L2 caches so we do not apply shared memory to this algorithm.

Algorithm 16 CPU parallel SSPTTM algorithm (CPU-SSPTTM).

Input: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with dense modes $D_{\mathcal{Y}}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, and an integer n;

Output: A semi-sparse tensor $\mathcal{Z} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \cdots \times I_N}$;

1: n^Z_{chunks}: the number of chunks in $\mathcal{Z}$
2: s^Y_{chunk}: the size of a chunk in $\mathcal{Y}$
3: s^Z_{chunk}: the size of a chunk in $\mathcal{Z}$
4: c_{ptr}: the beginnings of each chunk of $\mathcal{Y}$, size n^Z_{chunks}.
5: **parfor** $i = 0, \ldots, n^Z_{\text{chunks}} - 1$ **do**
6: **for** $j = c_{\text{ptr}}(i), \ldots, c_{\text{ptr}}(i+1) - 1$ **do**
7: $r = \text{inds}^Y_n(j)$;
8: **for** $c = 0, \ldots, R-1$ **do**
9: **for** $k = 0, \ldots, s^Y_{\text{chunk}} - 1$ **do**
10: $\text{val}^Z(i \times s^Z_{\text{chunk}} + r \times s^Y_{\text{chunk}} + k) += \text{val}^Y(j \times s^Y_{\text{chunk}} + k) \times u(r, c)$;
11: **end parfor**
12: **Return** $\mathcal{Z}$;

7.5 Sparse Tucker decomposition

Based on our optimizations of SPTTM and SSPTTM, we design the HOOI algorithm of Tucker decomposition (described in § 2.3.2) for a heterogeneous CPU-GPU platform.

7.5.1 TTM-Chain

The TTM-chain for a sparse input tensor consists of two types of TTMs—SPTTM and SSPTTM. The first TTM in Line 4 of Algorithm 2 is a SPTTM, and the following TTMs are all SSPTTMs. We use our fastest CUDA implementations of them to accelerate the TTM-chains. The output tensor of every TTM (either SPTTM or SSPTTM) resides in GPU memory, thus no memory transfer is needed inside a TTM-chain.

7.5.2 SVD

According to the property in § 3.2.2, the output $\mathcal{Y}$ of TTM-chain is dense in all except mode n. We treat it as a dense tensor, then $\mathbf{Y}_{(n)}$ is a dense matrix. For the SVD operation, we employ svd function ("`sgesvd`") from OpenBLAS library [186] on CPUs, while use SVD function ("`cusolverDnSgesvd`") from cuSOLVER [2]. However, from our experiments, the SVD function of cuSOLVER is not as efficient as OpenBLAS version, mainly because of the irregular shape of the matricized $\mathbf{Y}_{(n)}$.

7.6 Experiments

We test our algorithms on three platforms, one Intel multicore CPU platform and two GPU platforms. Our SPTTM performance is compared with state-of-the-art Tensor Toolbox [16] library and a recent sparse TTM implementation [107] on GPUs. We also analyze our GPU optimization methods incrementally and the performance behavior by varying product modes and rank sizes.

7.6.1 Platforms and Dataset

We use Intel Xeon CPU E5-2650 v4 and NVIDIA Tesla K40c and P100 platforms, their configurations are shown in table 7.2. NVIDIA Tesla K40c and P100 have much higher peak floating-point performance and memory bandwidth than the Intel Xeon CPU. All experiments perform single-precision floating-point values and the performance numbers are averaged over five runs. Without further specification, R is set to 16.

We use third-order and fourth-order sparse tensors from real applications, collected in FROSTT [151] and HaTen2 [78]. Tensors consist of medical data from Children's Healthcare of Atlanta project (choa with patient-medication-diagnosis), Never Ending Language Learning (NELL) project [36] (nell1 and nell2 with noun-verb-noun), Freebase RDF data [78] "freebase-music" and "freebase-sampled" (fb-m and fb-s with entity-entity-relation),

Table 7.2: Experimental Platforms Configuration

Parameters	Intel Xeon CPU E5-2650 v4	NVIDIA Tesla K40c	NVIDIA Tesla P100
Microarchitecture	Broadwell-EP	Kepler	Pascal
Frequency	2.2 GHz	0.75 GHz	0.72 GHz
#Physical cores	12	2880	3584
Peak SP Performance	845 GFlop/s	4290 GFlop/s	9300 GFlop/s
Last-level cache	30 MB	1.6 MB	4 MB
Memory size	792 GB	12 GB	16 GB
Memory bandwidth	77 GB/s	288 GB/s	732 GB/s
Compiler	gcc 5.4.1	nvcc 9.0	nvcc 9.0

data crawled from tagging systems [64] (deli with user-item-tag and deli4d with user-item-tag-date), Enron emails (enron with sender-receiver-word-date), Uber pickup data in 2016 (uber with date-hour-latitude-longitude), NIPS papers published from 1987 to 2003 (nips with id-author-word-year), and tags from Flickr (flickr with user-item-tag-date). The tensor property is shown in table 7.3. Please refer to the original dataset [78, 151] for further information.

Table 7.3: Description of sparse tensors.

Tensors	Order	Mode sizes	NNZ	Density
choa	3	$712K \times 10K \times 767$	27M	5.0×10^{-06}
nell2	3	$12K \times 9K \times 29K$	77M	1.3×10^{-05}
fb-m	3	$23M \times 23M \times 166$	100M	1.1×10^{-09}
fb-s	3	$39 \times 39 \times 532$	140M	1.7×10^{-10}
deli	3	$533K \times 17M \times 2M$	140M	6.1×10^{-12}
nell1	3	$3M \times 2M \times 26M$	144M	9.1×10^{-13}
nips	4	$2K \times 3K \times 14K \times 17$	3M	1.8×10^{-06}
uber	4	$183 \times 24 \times 1140 \times 1717$	3M	3.8×10^{-04}
enron	4	$6K \times 6K \times 244K \times 1K$	54M	5.5×10^{-09}
flickr	4	$320K \times 28M \times 2M \times 731$	113M	1.1×10^{-14}
deli4d	4	$533K \times 17M \times 2M \times 1K$	140M	4.3×10^{-15}

7.6.2 Overall Performance

We first show the speedups of our GPU-SPTTM algorithms over the OpenMP parallelized CPU-SPTTM in figure 7.2 and compare their performance to FCOO-SPTTM performance [107] on the GPU P100 platform [1] . The speedup numbers of each tensor are averaged among all its modes. SPTTM GPU performance is shown using the best one of all the five GPU implementations in § 7.3. GPU-SPTTM achieves up to $67\times$ speedups over CPU-SPTTM. Compared to the state-of-the-art work [107], our GPU-SPTTM implementations overperform FCOO-SPTTM by up to $3.9\times$. From our experiments, the best speedup of GPU-SPTTM is mostly obtained by GPU-SPTTM-SM (Algorithm 15), which verifies our optimizations. The detailed analysis on the five approaches will be given afterward. CPU-SPTTM's performance is obtained by using 12 threads.

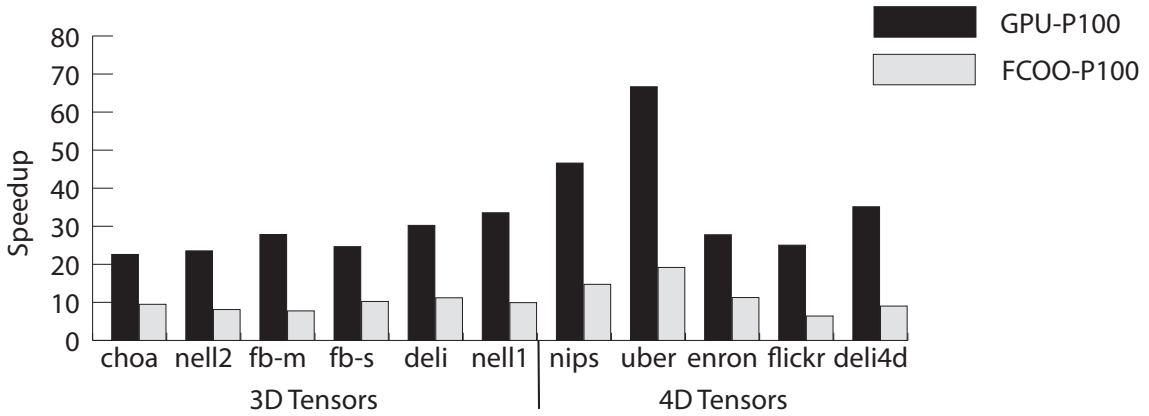


Figure 7.2: Our GPU-SPTTM and FCOO-SPTTM [107] speedups over CPU-SPTTM.

An interesting phenomenon is the speedup difference between tensor deli and deli4d. From table 4.3, deli4d has the same number of non-zeros with deli, but with an extra "date" mode, whereas GPU-SPTTM achieves higher speedup on the fourth-order deli4d. One main reason is that of the load imbalance between CUDA threads incurred by different fiber lengths of the input tensor $\mathcal{X}$. We calculate the standard deviation of fiber lengths for each

[1] We don't compare with Tensor Toolbox [16] and CTF [160] because they lack GPU parallel implementations for sparse tensors.

tensor mode. deli4d has a standard deviation 2.54×10^4 about a half magnitude less than deli's 1.21×10^5.

Figure 7.3 gives the actual SPTTM floating-point operations per second (flop/s) of "GPU-P100" and "GPU-K40c". "GPU-P100" achieves better performance, 16-51 GFlop/s, while "GPU-K40c" obtains less than 10 GFlop/s. The performance numbers obtained in this work on P100 is better than the FCOO-SPTTM [107] and sparse matrix-dense matrix multiplication [108]. However, compared to either the peak machine performance or the attainable performance limited by memory bandwidth, our performance is far below these bounds. There is space for further performance tuning.

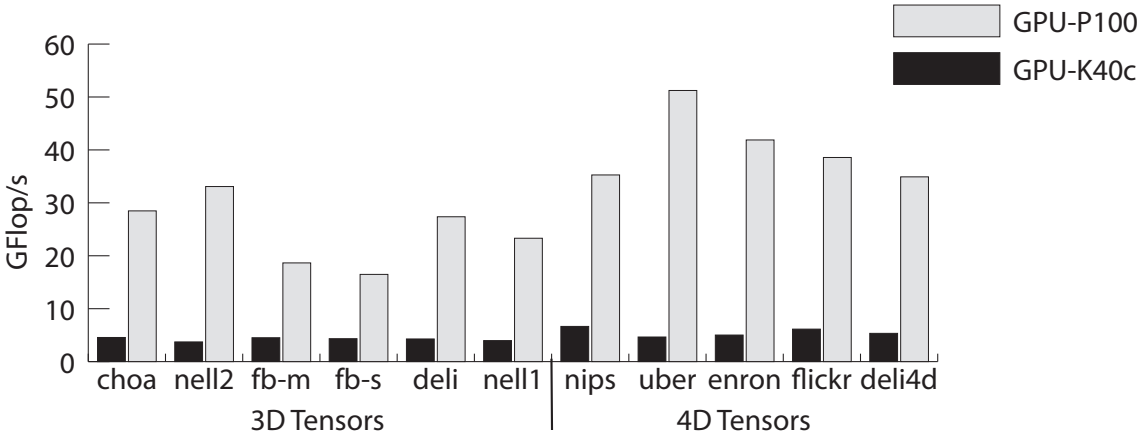


Figure 7.3: GPU-SPTTM performance in GFlop/s.

7.6.3 Analysis

This work is analyzed from different aspects: incremental GPU optimization effects, sequential algorithm comparison to show the advantage of our SPTTM algorithm, SSPTTM and SPTTM comparison for semi-sparse tensors, mode behavior, and rank behavior. After all, we apply our optimized algorithms to Tucker decomposition.

GPU Optimization Comparison

We analyze the five GPU SPTTM approaches in § 7.3: naïve implementation, fine thread granularity, coalesced memory access, rank blocking, and using shared memory, on a GPU P100 in figure 7.4. The times are normalized to the naïve GPU implementation and are averaged over all modes for every tensor. As the optimization methods incrementally applied to SPTTM, we tend to get shorter execution time. Naïve SPTTM is the simplest and slowest GPU implementation. SPTTM performance is incrementally improved: 53% by integrating fine thread granularity, 29% by arranging coalesced memory access, 7% by rank blocking, and 40% by using shared memory for writable data, on average of all tensors. Therefore, fine thread granularity and using shared memory optimizations are the two most effective optimizations for SPTTM, arranging coalesced memory access also improves some performance, while the effect of rank blocking depends on the given sparse tensors. We also test these approaches by setting R to 32. SPTTM performance is incrementally improved: 74% (+FG), 37% (+MC), -20% (+RB), and 43% (+SM), on average of all tensors. This shows rank blocking is more beneficial for small ranks, because of the re-loads of index ks and values, which coincides with our analysis in § 7.3.

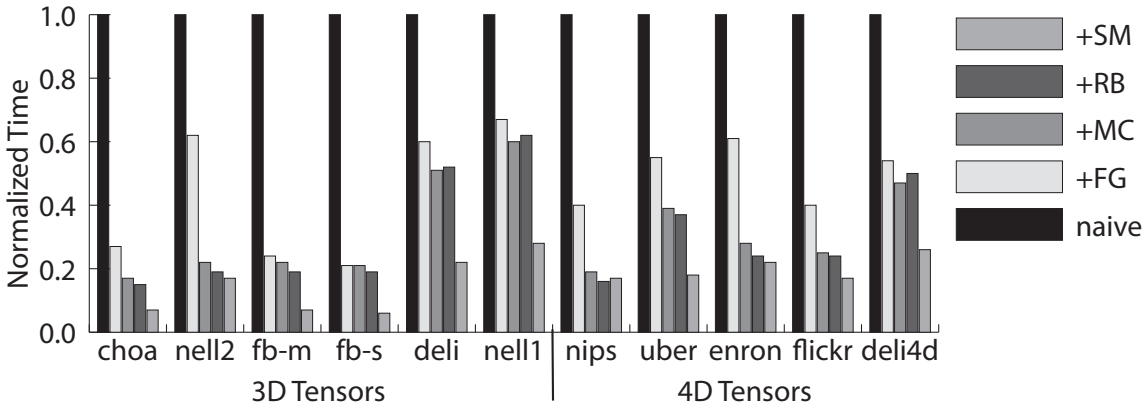


Figure 7.4: GPU optimization methods comparison on GPU P100.

Sequential SPTTM *Comparison*

We test the SPTTM from TENSOR TOOLBOX [16] on only three small tensors without exceeding memory. [2] Our SEQ-SPTTM achieves $3 - 120\times$ speedup over Tensor Toolbox. One reason is Tensor Toolbox is built on MATLAB environment, this may generate some extra overhead. For the fourth-order tensor nips, though it only has 3 million nonzeros, Tensor Toolbox runs much slower than we expect. This shows Tensor Toolbox may be not friendly enough for higher-order tensors.

Table 7.4: Sequential SPTTM performance comparison.

Time (sec)	Tensor Toolbox	SEQ-SPTTM
choa	7.39	2.78
nell2	86.50	7.33
nips	120.07	0.58

Table 7.5 shows the storage of Tensor Toolbox compared to our SPTTM in mode 1. TENSOR TOOLBOX consumes about twice storage space than SPTTM.

Table 7.5: Total storage (GBytes) of sparse tensors.

Tensors	choa	nell2	fb-m	fb-s	deli	nell1	nips	uber	enron	flickr	deli4d
TTBox	2.72	2.30	30.35	44.90	20.69	11.80	0.40	0.09	1.77	13.81	46.07
SPTTM	1.55	1.32	17.34*	25.65*	11.82	6.74	0.23	0.05	1.01	7.89	19.75*

* Since an TTM only needs one index instead of all, these tensors can fit into GPU memory.

SSPTTM *v.s.* SPTTM *for Semi-Sparse Tensors*

We compare the performance of GPU-SSPTTM with GPU-SPTTM for semi-sparse tensors in figure 7.5. We use the second TTM operation in the TTM-chain of Tucker decomposition (Algorithm 2) to ensure the input tensor is semi-sparse. SPTTM handles the semi-sparse input tensor as a general sparse tensor. Since the input semi-sparse tensors could be easily as large as the tensors in table 7.3 because of the introduced dense mode. We can only run

[2] Since Cyclops Tensor Framework (CTF) [160, 161] indexes a sparse tensor using a one-dimensional long index to help distribute nonzero elements in a certain way in distributed platforms, all tensors we tested cannot be represented after vectorization, so they cannot run using CTF except in distributed memory environment.

SPTTM for this operation on small tensors. GPU-SSPTTM achieves up to $74\times$ and $12\times$ speedups on GPU P100 and K40c platforms respectively. Compared to SPTTM, which treats semi-sparse tensors as general sparse tensors, SSPTTM achieves up to $4.5\times$ speedup. This figure shows the usefulness of the SSPTTM algorithm for semi-sparse tensors.

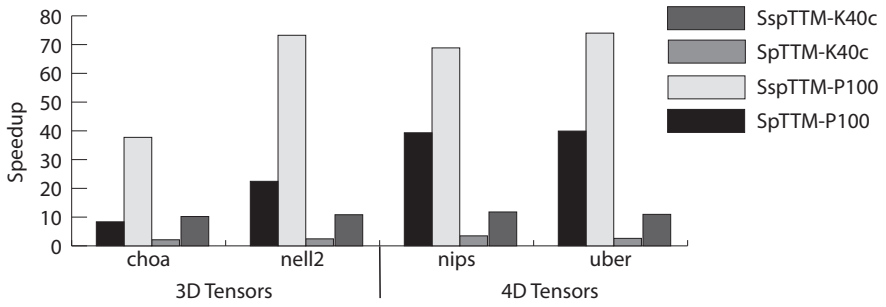


Figure 7.5: GPU-SPTTM and GPU-SSPTTM speedups over corresponding OpenMP parallelized CPU implementations on two NVIDIA GPU platforms.

Mode behavior

We compare the "GPU-P100" SPTTM behavior in different modes, using the best execution time of the five approaches. Figure 7.6 takes mode-1 SPTTM as the baseline, then computes the SPTTMs in other modes normalized to it. Some tensors have very diverse SPTTM performance in different modes. From our observations, the modes in which fibers' length has a small standard deviation tend to be faster compared to other modes. For example, tensor choa has small fiber sizes in mode 2 and 3, so their SPTTM performance is much better than that in mode 1.

Rank behavior

Figure 7.7 shows the relative performance of mode-1 "GPU-P100" SPTTM on tensors choa, nell2, uber, and enron by increasing the rank-size. We only test on small tensors because when rank-size is 32 or 64, some tensors are too large to reside in GPU memory. As the rank increases, the performance increases from R is 8 to 16 but stops increasing after 16.

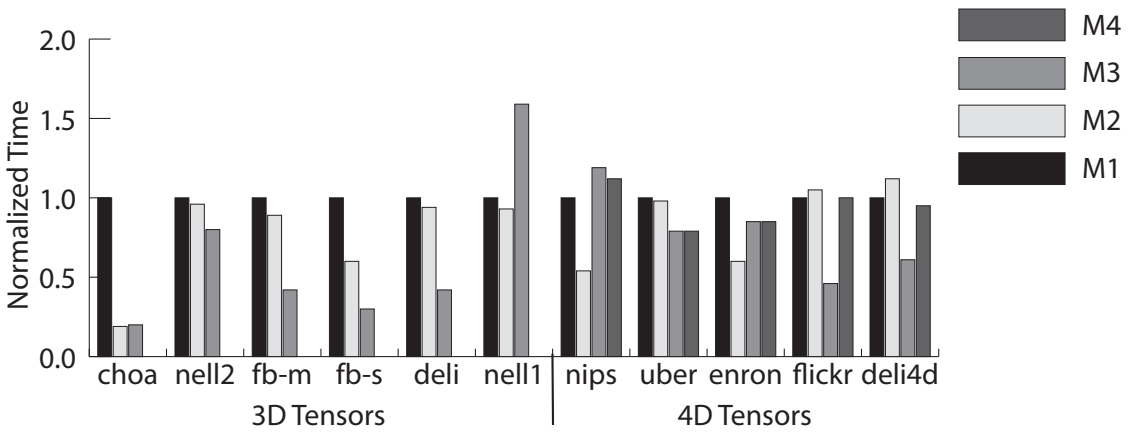


Figure 7.6: Relative SPTTM time in different modes.

This is because our algorithms are able to fully use the factor rows when $R = 16$.

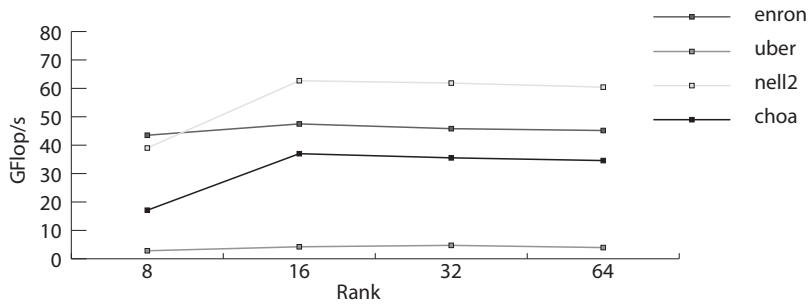


Figure 7.7: Execution time of small tensors in different rank sizes.

Tucker Decomposition

Table 7.6 shows the execution time of the Tucker decomposition by applying our optimized SPTTM and SSPTTM algorithms. Due to space limitation, large tensors cannot run on GPU platforms due to the non-trivial intermediate results. The optimized Tucker decomposition achieves up to $3.2\times$ speedup. From Tucker decompositions, we observed that SVD and TTM-chain performance are both important to Tucker performance. We test SVD by using OpenBLAS and cuSOLVER. However, SVD of cuSOLVER cannot achieve as good performance as OpenBLAS, especially for truncated SVD. Therefore, we use OpenBLAS to solve SVD and include the memory transfer time into our GPU Tucker implementation.

Table 7.6: Tucker decomposition performance.

	CPU (sec)	GPU (sec)
choa	49.3	15.6
nell2	48.6	16.7
enron	389.6	290.3

7.7 Summary

The key algorithmic principle in this chapter is avoiding data transformation, just as it was in Chapter 6. Combined with multicore and GPU optimizations—which include employing fine thread granularity, coalescing memory accesses, rank blocking, and using local (fast) memory ("shared memory" on GPUs)—yield sequential SPTTM is $3 - 120\times$ faster than the SPTTM in TENSOR TOOLBOX, and GPU parallel algorithms that, on NVIDIA P100, improve over our OpenMP-parallelized CPU-SPTTM and other state-of-the-art GPU implementations by $3.9\times$. From our analysis, different input sparse tensors, ranks, and operating on different modes all influence SPTTM performance. Looking forward, new techniques to select the adaption parameters of the SPTTM algorithms are needed.

CHAPTER 8

PARTI! LIBRARY DESIGN AND IMPLEMENTATION

8.1 PARTI! Design and Implementation

PARTI!, a parallel tensor infrastructure, supports fundamental tensor operations and tensor decompositions emphasizing on sparse data. Multiple platforms are supported including multicore CPUs, GPUs, distributed memory systems, as well as a new prototype platform, the Emu [58].[1] PARTI! is written in C language, with OpenMP, CUDA, MPI, and Cilk (for Emu) supports for parallel implementations, and has MATLAB interface for application users.

The tensor methods supported in PARTI! are listed in table 8.1. It includes methods for queries, transformations, fundamental products, factorizations, and their constraints. The support for high-order tensor factorizations, like tensor train and hierarchical Tucker, and incorporating constraints are left for future work.

The platforms supported in PARTI! are shown in table 8.2 for the most popular COO

[1] Emu is a highly scalable near memory architecture with migrating threads especially for data intensive problems.

Table 8.1: PARTI! supported Tensor methods. To be supported methods are shown in gray.

Category	Tensor methods
Queries	fiber, slice, sub-tensor
Transformations	matricization (unfolding or flattening), reshape, tensorization
Fundamental products	element-wise products, mode-n tensor-matrix/vector product, contraction matricized tensor times Khatri-Rao product (MTTKRP) outer product, Kronecker product, Khatri-Rao product
Factorizations	*Low-order*: CP, Tucker, ... *High-order*: tensor train, hierarchical Tucker, ...
Constraints	nonnegativity, orthogonality, sparsity

sparse tensor format and our newly proposed HICOO format in Chapter 4. Except Tucker decomposition for HICOO format, all cases are supported for multicore CPUs and GPUs. CP decomposition is implemented on all the four platforms for COO format and HICOO format except Emu platform. Part of our future work is to fill up the rest functionality for PARTI!. The sections describing the corresponding functionalities are also shown in table 8.2.

8.1.1 COO-CPD on Multicore CPUs

Sequential MTTKRP and CPD algorithms based on COO format in PARTI! are implemented following the algorithms in TENSOR TOOLBOX [16]. The only difference is that PARTI! uses row-by-row iteration and row-major layout for dense factor matrices, while TENSOR TOOLBOX uses column-by-column iteration and column-major layout. PARTI! COO-MTTKRP and CPD implementations are much faster than TENSOR TOOLBOX, one is because of the usage of the faster C language; the other is because of the row-major pattern which makes a better memory reuse for a sparse tensor in a MTTKRP operation.

COO-MTTKRP is parallelized using OpenMP using two parallel strategies: locking and privatization (as described in Section 4.3). These strategies are different ways to avoid data races during matrix updates in MTTKRP (Section 4.3). In the locking approach, we allocate one lock per matrix row to guard against concurrent updates. In the privatization method, we use thread-local buffers for the whole output matrix. Each thread first updates their local buffers, then uses a global reduction operation to get the final updated matrix. Privatization accelerates COO-MTTKRP at a cost of extra space to save local copies. For multicore parallel COO-CPD, we support high-performance math libraries, such as the BLAS [24], OpenBLAS [186], Intel MKL [48], and LAPACK [7, 8], wherever parallel dense matrix operations are needed.

Table 8.2: PARTI! supported tensor formats, factorizations and platforms.

Tensor formats	Factorizations	Multicore CPUs	GPUs	Distributed systems	Emu
COO	CP	Y (§ 8.1.1)	Y (§ 8.1.2)	Y (§ 8.1.3)	Y (§ 8.1.4)
	Tucker	Y (§ 7)	Y (§ 7)	N	N
HICOO	CP	Y (§ 4)	Y (§ 8.1.5)	Y (§ 8.1.6)	N
	Tucker	N	N	N	N

8.1.2 COO-CPD on GPUs

COO-MTTKRP on GPUs uses the same optimization approaches with sparse COO-TTM (§ 7.3), including naïve implementation, employing fine thread granularity, arranging coalesced memory access, and rank blocking. For COO-MTTKRP, it is hard to take the advantage of the GPU shared memory. All the implementations use atomic operations from CUDA to solve the data race problem. From our experiments, the best implementation is arranging coalesced memory access or rank blocking for different tensors (also observed in TTM in § 7.6).

8.1.3 COO-CPD on Distributed Systems

We use medium-grained and coarse-grained distribution strategies for COO-CPD.

For a given mode, the coarse-grained strategy distributes a sparse tensor by assigning every slice in its entirety (all its non-zero elements) to a processor [41]. The factor matrices are assigned correspondingly, i.e., each processor is assigned to the matrix rows it needs to complete the CPD. In this way, voluminous duplication of factor matrices exists on processors. Therefore, synchronizing communications are necessary. Coarse-grained strategy distributes a sparse tensor and factor matrices differently for an MTTKRP in a different mode. Thus, the tensor and matrices are all stored N times.

The medium-grained strategy, proposed by Shaden et al. [153], distributes a sparse tensor into sub-tensors along with all tensor modes. Factor matrices are also split according to the index ranges of the sub-tensor on each processor. A sparse tensor and factor matrices

is stored only once for different MTTKRPs. The detailed comparison between these two strategies can be found in a recent work by Chakaravarthy et al. [38].

Regardless of the distribution strategy, every processor runs a sequential COO-MTTKRP and CPD to complete a local computation. We use MPI to implement distributed COO-CPD. All communication relies on MPI collectives.

8.1.4 COO-CPD on Emu

We also extend COO-MTTKRP to the new Emu architecture [58, 74]. The Emu Chick prototype consists of eight nodelets, each with four Gossamer Cores (GCs). We need to carefully allocate data on Emu to maximally reduce the thread migrating between nodelets. Currently, we use a coarse-grained-like strategy to distribute a sparse tensor and factor matrices. The programming model for Emu is based on Cilk [26]. COO-MTTKRP are parallelized with *cilk_spawn*, *cilk_sync*, and *cilk_for* on Emu.

8.1.5 HICOO-CPD on GPUs

HICOO-MTTKRP on GPUs uses the same optimization approaches with COO-MTTKRP (§ 8.1.2), including naïve implementation, employing fine thread granularity, arranging coalesced memory access, rank blocking, and using shared memory as an extra. The GPU shared memory is also used to save blocked factor matrices to be updated. From our experiments, the best implementation is coalesced memory access, rank blocking, or using shared memory for different tensors. We envision additional optimizations for HICOO-MTTKRP on GPUs, in particular, to address load balancing issues, perhaps by a "mixed-grain" nonzero partitioning strategy.

8.1.6 HICOO-CPD on Distributed Systems

We use the medium-grained strategy for a distributed HICOO-MTTKRP and CPD. Two CPD algorithms are implemented, the CP-ALS (Section 2.3.1) and CP-APR algorithms [40].

The CANDECOMP/PARAFAC Alternating Poisson Regression (CP-APR) algorithm applies to sparse count data assuming a Poisson distribution and is based on majorization-minimization approach. An MTTKRP-like operation is the most computational expensive step of CP-APR. HICOO format is used for the input sparse tensor to improve data locality and the performance of CP-ALS and CP-APR.

8.2 Summary of Standard Library Interfaces

The capabilities of tensor libraries vary widely, and many libraries focus on a particular application domain. Some features see very little support to date, including sparse tensor support, parallelism, and tensor network methods, largely because these are still open research questions. In table 8.3 we roughly break down the features of different packages according criteria such as supported computational cores, decomposition methods, and tensor features, where supported features are marked by a 'Y.' The packages we choose are very popular at one time or keep updated till recently. Note that some implementations, such as HyperTensor [85, 149], have not released their code yet, so they are not included in the software list. Due to space limit, some packages are also not included, such as vmmlib [18, 19], Tensors [61, 132, 133] since they are not actively maintained.

The first column denotes whether elementwise operations are supported. The following columns denote whether general contractions, TTM, or MTTKRP are supported, respectively. We consider a package supports MTTKRP when implemented as a whole. The decomposition columns correspond to whether CP and Tucker decompositions introduced in § 2.3 are natively supported along with two higher-order tensor decompositions: hierarchical Tucker (HT) and tensor train (TT). Note that this is a very high-level overview and does not indicate whether methods are optimized, or whether the factorizations can be used as a native data type.

We also show the maximum supported order of a tensor for every package, whether a native dense or sparse data type is supported, and whether parallelism is taken advan-

tage of. The type of parallelism used – shared-memory, distributed-memory, or accelerator platforms – is denoted by superscripts. Finally we describe application domain and programming environment. (Factorization frameworks clearly favor MATLAB!). Further information about each package can be found in the associated references.

In the last four high performance packages (GenTen, SPLATT, vmmlib, and PARTI!) which support parallelism for tensor decompositions, PARTI! has the most widely support for tensor operations, tensor decompositions, and parallelism.

Table 8.3: Capabilities of tensor frameworks.

Software	Computational Cores				Tensor Decomposition				Tensor Features				Software Information	
	$\mathcal{A} * \mathcal{B}$	$\mathcal{A} \times_n^m \mathcal{B}$	$\mathcal{A} \times_n \mathrm{B}$	MTTKRP	CP	Tucker	HT	TT	Order	Dense	Sparse	Parallel	Domain	Environ.
Basic Functionality														
Numpy [47, 124, 167]	Y	N	N	N	N	N	N	N	Arb.	Y	N	N	Gen.	Python
tensor (Theano) [22, 23]	Y	Y	Y	N	N	N	N	N	Arb.	Y	N	N	Gen.	Python
TCE (NWChem) [10, 11]	Y	Y	Y	N	N	N	N	N	≤ 4	Y	Y	Y[2,3]	Chem.	DSL
libtensor [59, 60]	Y	Y	Y	N	N	N	N	N	Arb.	Y	Y	Y[1]	Gen.	C
Boost.MultiArray [62, 63]	N	N	N	N	N	N	N	N	Arb.	Y	N	N	Gen.	C++
FTensor [96, 97]	Y	Y	Y	N	N	N	N	N	≤ 4	Y	N	N	Gen.	C++
ltensor [106]	Y	Y	Y	N	N	N	N	N	≤ 4	Y	N	N	Gen.	C++
Blitz++ [170]	Y	Y	Y	N	N	N	N	N	≤ 11	Y	N	N	Gen.	C++
TiledArray [34, 35]	Y	Y	Y	N	N	N	N	N	Arb.	Y	Y	Y[1,2]	Gen.	C++
BTAS [166]	Y	Y	Y	N	N	N	N	N	Arb.	Y	N	N	Gen.	C++
MShadow [39]	Y	Y	Y	N	N	N	N	N	Arb.	Y	N	Y[3]	ML	C++
Uni10 [81, 121]	Y	Y	Y	N	N	N	N	N	Arb.	Y	N	Y[3]	Gen.	C++
TL (Dynare++) [57, 79]	Y	Y	Y	N	N	N	N	N	Arb.	Y	Y	Y[1]	FIN	C++
CTF [160, 163]	Y	Y	Y	N	N	N	N	N	Arb.	Y	Y	Y[1,2,3]	Gen.	C++
Redberry [27, 28]	Y	Y	Y	N	N	N	N	N	Arb.	Y	N	N	Gen.	JAVA
Tensor Decompositions														
Tensor Toolbox [14, 16]	Y	Y	Y	Y	Y	Y	N	N	Arb.	Y	Y	N	Gen.	MATLAB
Tensorlab [171]	Y	Y	Y	Y	Y	N	N	Y	Arb.	Y	Y	Y[1]	Gen.	MATLAB
N-way Toolbox [9, 29]	Y	N	Y	N	Y	Y	N	N	Arb.	Y	N	N	Gen.	MATLAB
PLS_Toolbox [179]	Y	Y	Y	N	Y	Y	N	N	Arb.	Y	Y	N	Chem.	MATLAB
TT-Toolbox [125]	Y	Y	Y	N	N	Y	N	Y	Arb.	Y	Y	N	Gen.	MATLAB
H-Tucker [94, 95]	Y	Y	Y	N	N	N	Y	Y	Arb.	Y	Y	N	Gen.	MATLAB
CuBatch [67, 68]	Y	Y	Y	N	Y	Y	N	N	Arb.	Y	N	N	Gen.	MATLAB
TDALAB [188, 189]	Y	N	N	N	Y	Y	N	N	Arb.	Y	Y	N	Gen.	MATLAB
TENSORBOX [136]	Y	Y	Y	Y	Y	Y	N	N	Arb.	Y	Y	N	Gen.	MATLAB
TensorLy [92]	Y	Y	Y	N	Y	Y	N	N	Arb.	Y	N	Y[1,3]	Gen.	Python
scikit-tensor [122]	Y	N	Y	N	Y	Y	N	N	Arb.	Y	Y	N	Gen.	Python
PyTensor [144, 145, 183]	N	N	Y	N	N	Y	N	N	Arb.	Y	Y	N	Sim.	Python
HaTen2 [77, 78]	N	Y	Y	Y	Y	Y	N	N	≤ 3	N	Y	Y[2]	ML	JAVA
ITensor [118]	Y	Y	Y	N	N	N	N	Y	Arb.	Y	Y	N	Sim.	C++
DFacTo [41, 42]	N	N	N	Y	Y	N	N	N	≤ 3	N	Y	Y[1]	Gen.	C++
GenTen [138]	N	N	N	Y	Y	N	N	N	Arb.	N	Y	Y[1,2,3]	Gen.	C++
SPLATT [156, 157]	N	N	Y	Y	Y	Y	N	N	Arb.	N	Y	Y[1,2,3]	Gen.	C
PARTI! [102]	Y	N	Y	Y	Y	Y	N	N	Arb.	N	Y	Y[1,2,3]	Gen.	C

MTTKRP: Optimized MTTKRP; **Order**: Arb. - Arbitrary; **Environ.**: DSL - Domain Specific Language;
Parallelized: [1]Multi-Threading Parallelism; [2]Distributed Parallelism; [3]Accelerator Parallelism (GPUs, Xeon Phi processors)
Domain: Gen. - General; Sim. - Simulation; Chem. - Chemistry; ML - Machine Learning; FIN - Finance;

CHAPTER 9
CONCLUSION AND FUTURE DIRECTIONS

9.1 Conclusion

This dissertation develops high-performance tensor algorithms for large-scale dense and sparse data by optimizing them from diverse scopes. We summarize the main results of the whole dissertation to give a broad idea of the efficiency of these optimization approaches.

- Memoization (Chapter 3): By using memoization in sparse MTTKRP and CPD algorithms, ADATM outperforms the state-of-the-art library SPLATT [157] by up to $8\times$, on real sparse tensors with orders as high as 85.

- HICOO (Chapter 4): HICOO achieves an average of $6.8\times$ speedup over COO from PARTI! [102] by using up to $2.5\times$ less storage than it and $3.1\times$ speedup over CSF from SPLATT [157] by using comparable storage with only one CSF representation, which are the two state-of-the-art libraries. Our reordering methods obtain up to $2.67\times$ further speedups.

- In-place dense approach (Chapter 6): Our INTENSLI-generated TTM codes outperform the TTM implementations available in two widely used tools, the TENSOR TOOLBOX [89] and CTF [162], by about $4\times$ and $13\times$, respectively.

- In-place sparse approach (Chapter 7): Our sparse TTM on GPUs is $3.9\times$ faster than the state-of-the-art GPU implementation [107]. Our Tucker decomposition on GPUs outperforms CPU Tucker decomposition by $3.2\times$.

- PARTI! library (Chapter 8): PARTI! implements diverse CP and Tucker decompositions on multiple platforms including multicore CPUs, GPUs, distributed systems, and Emu, supporting two sparse tensor formats, the popular COO and our HICOO.

To conclude this dissertation, memoization and in-place techniques are useful and applicable to various tensor decompositions and baseline tensor operations; HICOO format is an efficient way to compress a sparse tensor and achieves good data locality and performance speedup for HICOO based tensor algorithms. Tuning the algorithmic parameters enclosed in these approaches is critical for their performance. Besides, flexible parameters are necessities for evolving computer architectures. Some tuning methods like model-driven mechanism are already used in our work, while more intelligent and automatic performance tuning is still in a dire need. Automatic performance tuning of tensor methods considering different input tensors and diverse computer architectures with a fast search in their parameter space is one direction worth to explore in the future. Another direction is to develop a fast estimation of the actual parameter values, e.g., for HICOO format.

9.2 Future Directions

9.2.1 CP Decomposition

Looking forward to our ADATM work, it is worth to apply our adaptive tensor memoization algorithm to our newly proposed HICOO format-based tensor decompositions. We also believe a closer inspection of not just the arithmetic but also communication properties of our method, coupled with more architecture-specific tuning, manycore co-processor acceleration, and extensions for distributed memory, are ripe opportunities for future work.

The future directions for HICOO format will be exploring the behavior of the performance critical parameters of HICOO and predict their optimal choice by sampling nonzero distribution of an input tensor. For tensor reordering, further acceleration of reordering process is needed, such as parallelizing this process and merging HICOO construction and reorder processes together. We also consider alternative reordering approaches, such as co-clustering techniques. It is also important to develop HICOO variants of other widely used kernels, such as tensor-times-matrix multiplication (TTM).

9.2.2 Tucker Decomposition

One direction is to show the impact of our performance improvements of INTENSLI in the context of higher-level decomposition algorithms, such as Tucker, hierarchical Tucker, or tensor trains, among others [69, 73, 126, 165]. The case of dense tensors has numerous scientific applications, including, for instance, time series analysis for molecular dynamics [143], which we will pursue.

The sparse Tucker decomposition still has a large optimization space: first, using HICOO format to obtain better data locality; second, overlapping some preprocessing stages and memory transfer with the TTM computation time and better handling the load-balance issue on GPUs; third, extending our algorithms to multi-GPU platforms to support larger sparse tensors.

9.2.3 Higher-Order Tensor Decompositions

Many interesting topics can be explored from higher-order tensor decompositions such as tensor train, hierarchical Tucker decompositions, which are emerging popularized in applications. These higher-order tensor decompositions can solve the issue with both CP and Tucker decompositions that they may lead to a flattening of multi-way relationships. That is, CP and Tucker decompositions assume a model in which every mode interacts with the other modes but ignore the situations where modes may interact among themselves in subgroups or hierarchies. Developing efficient higher-order decompositions could be very useful to large-scale applications.

9.2.4 Develop Highly Hybrid Tensor Algorithms

State-of-the-art high-performance tensor algorithms are mostly emphasized on only one parallelism level, while few of them employed two parallelism levels, such as SPLATT [157] and CTF [162]. It is very promising to fully explore a machine's computational power of all kinds, including distributed systems, multicore CPUs, and accelerators (GPUs or Intel

Xeon Phi processors), to accelerate tensor methods. This topic involves load balance and task scheduling along with other interesting research problems. In this direction, maybe a more generic framework or a domain-specific language can be developed for not only tensor methods but more computational kernels. The tensor algebra compiler (TACO) [87] generates tensor algebra kernels including tensor, matrix, and vector operations, but it is not adaptive to multiple platforms yet.

REFERENCES

[1] CUDA C best practices guide.

[2] cuSOLVER v9.0.

[3] ACAR, E., HARRISON, R. J., OLKEN, F., ALTER, O., HELAL, M., OMBERG, L., BADER, B., KENNEDY, A., PARK, H., BAI, Z., KIM, D., PLEMMONS, R., BEYLKIN, G., KOLDA, T., RAGNARSSON, S., DELATHAUWER, L., LANGOU, J., PONNAPALLI, S. P., DHILLON, I., LIM, L.-H., RAMANUJAM, J. R., DING, C., MAHONEY, M., RAYNOLDS, J., ELDN, L., MARTIN, C., REGALIA, P., DRINEAS, P., MOHLENKAMP, M., FALOUTSOS, C., MORTON, J., SAVAS, B., FRIEDLAND, S., MULLIN, L., AND VAN LOAN, C. Future directions in tensor-based computation and modeling, 2009.

[4] ACAR, E., AND YENER, B. Unsupervised multiway data analysis: A literature survey. *Knowledge and Data Engineering, IEEE Transactions on 21*, 1 (Jan 2009), 6–20.

[5] AKBUDAK, K., AND AYKANAT, C. Exploiting locality in sparse matrix-matrix multiplication on many-core architectures. *IEEE Transactions on Parallel and Distributed Systems 28*, 8 (Aug 2017), 2258–2271.

[6] ANANDKUMAR, A., GE, R., HSU, D., KAKADE, S. M., AND TELGARSKY, M. Tensor decompositions for learning latent variable models. *J. Mach. Learn. Res. 15*, 1 (Jan. 2014), 2773–2832.

[7] ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, L., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY,

A., AND SORENSEN, D. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, 1999.

[8] ANDERSON, E., BAI, Z., DONGARRA, J., GREENBAUM, A., MCKENNEY, A., DU CROZ, J., HAMMARLING, S., DEMMEL, J., BISCHOF, C., AND SORENSEN, D. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing* (Los Alamitos, CA, USA, 1990), Supercomputing '90, IEEE Computer Society Press, pp. 2–11.

[9] ANDERSSON, C. A., AND BRO, R. The N-way Toolbox for MATLAB. *Chemometrics and Intelligent Laboratory Systems 52*, 1 (Aug 2000), 1–4.

[10] AUER, A. A., ET AL. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics 104*, 2 (2006), 211–228.

[11] AUER, A. A., ET AL. Tensor Contraction Engine in NWChem (Version 6.8). Available from `http://www.nwchem-sw.org`, Dec 2017.

[12] AZAD, A., BALLARD, G., BULUÇ, A., DEMMEL, J., GRIGORI, L., SCHWARTZ, O., TOLEDO, S., AND WILLIAMS, S. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing 38*, 6 (2016), C624–C651.

[13] AZAD, A., AND BULUÇ, A. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2017), pp. 688–697.

[14] BADER, B. W., AND KOLDA, T. G. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software 32*, 4 (December 2006), 635–653.

[15] BADER, B. W., AND KOLDA, T. G. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing 30*, 1 (December 2007), 205–231.

[16] BADER, B. W., KOLDA, T. G., ET AL. MATLAB Tensor Toolbox (Version 3.0-dev). Available online, Oct. 2017.

[17] BALLARD, G., CARSON, E., DEMMEL, J., HOEMMEN, M., KNIGHT, N., AND SCHWARTZ, O. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica 23* (2014), pp. 1–155.

[18] BALLESTER, R. vmmlib: A templatized C++ vector and matrix math library (Version 1.6.2). Available from `http://vmml.github.io/vmmlib/`, 2013.

[19] BALLESTER-RIPOLL, R., SUTER, S. K., AND PAJAROLA, R. Analysis of tensor approximation for compression-domain volume visualization. *Computers & Graphics 47* (2015), 34–47.

[20] BASKARAN, M., MEISTER, B., LETHIN, R., AND CAI, J. Optimization of symmetric tensor computations. In *IEEE Conference on High Performance Extreme Computing (HPEC), Waltham, MA, USA* (Sep 2015), IEEE, September.

[21] BASKARAN, M., MEISTER, B., VASILACHE, N., AND LETHIN, R. Efficient and scalable computations with sparse tensors. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on* (Sept 2012), pp. 1–6.

[22] BASTIEN, F., LAMBLIN, P., PASCANU, R., BERGSTRA, J., GOODFELLOW, I. J., BERGERON, A., BOUCHARD, N., AND BENGIO, Y. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[23] BASTIEN, F., LAMBLIN, P., PASCANU, R., BERGSTRA, J., GOODFELLOW, I. J., BERGERON, A., BOUCHARD, N., AND BENGIO, Y. Basic tensor functionality in theano (Version 1.0.2). Available from `https://github.com/Theano/Theano`, May 2018.

[24] BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETITET, A., POZO, R., REMINGTON, K., AND WHALEY, R. C. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software 28* (2001), 135–151.

[25] BLELLOCH, G. Scans as primitive parallel operations. *IEEE Trans. Comput. 38*, 11 (nov 1989), 1526–1538.

[26] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1995), PPOPP '95, ACM, pp. 207–216.

[27] BOLOTIN, D. A., AND POSLAVSKY, S. V. Introduction to Redberry: a computer algebra system designed for tensor manipulation. *CoRR abs/1302.1219* (2013).

[28] BOLOTIN, D. A., AND POSLAVSKY, S. V. Redberry: an open source computer algebra system designed for algebraic manipulations with tensors (Version 1.1.9). Available from `http://redberry.cc`, 2015.

[29] BRO, R., AND ANDERSSON, C. A. The N-way Toolbox (Version 3.31). Available from `http://www.models.life.ku.dk/nwaytoolbox`, July 2013.

[30] BULUÇ, A., FINEMAN, J. T., FRIGO, M., GILBERT, J. R., AND LEISERSON, C. E. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using

compressed sparse blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2009), SPAA '09, ACM, pp. 233–244.

[31] BULUÇ, A., AND GILBERT, J. R. Challenges and advances in parallel sparse matrix-matrix multiplication. In *2008 37th International Conference on Parallel Processing* (Sept 2008), pp. 503–510.

[32] BULUÇ, A., AND GILBERT, J. R. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing* (April 2008), pp. 1–11.

[33] BULUÇ, A., AND GILBERT, J. R. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing 34*, 4 (2012), C170–C191.

[34] CALVIN, J. A., LEWIS, C. A., AND VALEEV, E. F. Scalable task-based algorithm for multiplication of block-rank-sparse matrices. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms* (New York, NY, USA, 2015), IAˆ3 '15, ACM, pp. 4:1–4:8.

[35] CALVIN, J. A., AND VALEEV, E. F. TiledArray: A massively-parallel, block-sparse tensor framework (Version v0.6.0). Available from `https://github.com/valeevgroup/tiledarray`, Nov 2016.

[36] CARLSON, A., BETTERIDGE, J., KISIEL, B., SETTLES, B., HRUSCHKA, E., AND MITCHELL, T. Toward an architecture for never-ending language learning.

[37] CARROLL, J. D., AND CHANG, J.-J. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika 35*, 3 (Sep 1970), 283–319.

[38] CHAKARAVARTHY, V. T., CHOI, J. W., JOSEPH, D. J., MURALI, P., PANDIAN, S. S., SABHARWAL, Y., AND SREEDHAR, D. On optimizing distributed Tucker decomposition for sparse tensors. In *Proceedings of the 32nd ACM International Conference on Supercomputing* (2018), ICS '18.

[39] CHEN, T. Matrix Shadow: Lightweight CPU/GPU matrix and tensor template library in C++/CUDA for (deep) machine learning (Version 1.1). Available from `https://github.com/tqchen/mshadow`, Dec 2014.

[40] CHI, E. C., AND KOLDA, T. G. On tensors, sparsity, and nonnegative factorizations. *SIAM Journal on Matrix Analysis and Applications 33*, 4 (2012), 1272–1299.

[41] CHOI, J. H., AND VISHWANATHAN, S. DFacTo: Distributed factorization of tensors. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 1296–1304.

[42] CHOI, J. H., AND VISHWANATHAN, S. DFacTo: Distributed factorization of tensors (Initial Version). Available from `http://web.ics.purdue.edu/~choi240/`, 2014.

[43] CHOI, J. W., SINGH, A., AND VUDUC, R. W. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 115–126.

[44] CICHOCKI, A. Era of big data processing: A new approach via tensor networks and tensor decompositions. *CoRR abs/1403.2048* (2014).

[45] CICHOCKI, A., LEE, N., OSELEDETS, I. V., PHAN, A., ZHAO, Q., AND MANDIC, D. Low-rank tensor networks for dimensionality reduction and large-scale optimization problems: Perspectives and challenges part 1. *ArXiv e-prints* (Sept. 2016).

[46] COMON, P. Tensors: a brief introduction. *IEEE Signal Processing Magazine 31*, 3 (May 2014), 44–53.

[47] CONTRIBUTORS, N. P. NumPy (Version 1.14). Available from `http://www.numpy.org`, Jan 2018.

[48] CORPORATION, I. Intel math kernel library. Tech. rep., 2018.

[49] COUTINHO, A. L. G. A., MARTINS, M. A. D., SYDENSTRICKER, R. M., AND ELIAS, R. N. Performance comparison of data-reordering algorithms for sparse matrixvector multiplication in edge-based unstructured grid computations. *International Journal for Numerical Methods in Engineering 66*, 3 (2006), 431–460.

[50] DE LATHAUWER, L. Decompositions of a higher-order tensor in block terms — part II: Definitions and uniqueness. *SIAM J. Matrix Anal. Appl 30*, 3 (2008), 1033–1066.

[51] DE LATHAUWER, L. Decompositions of a higher-order tensor in block terms—part I: Lemmas for partitioned matrices. *SIAM Journal on Matrix Analysis and Applications 30*, 3 (2008), 1022–1032.

[52] DE LATHAUWER, L., DE MOOR, B., AND VANDEWALLE, J. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl 21* (2000), 1253–1278.

[53] DE LATHAUWER, L., DE MOOR, B., AND VANDEWALLE, J. On the best rank-1 and Rank-(R1,R2,. . .,RN) approximation of higher-order tensors. *SIAM J. Matrix Anal. Appl. 21*, 4 (Mar. 2000), 1324–1342.

[54] DE LATHAUWER, L., AND NION, D. Decompositions of a higher-order tensor in block termspart III: Alternating least squares algorithms. *SIAM Journal on Matrix Analysis and Applications 30*, 3 (2008), 1067–1083.

[55] DE LATHAUWER, L., VERVLIET, N., BOUSS, M., AND DEBALS, O. Dealing with curse and blessing of dimensionality through tensor decompositions, 2017.

[56] DI NAPOLI, E., FABREGAT-TRAVER, D., QUINTANA-ORT, G., AND BIENTINESI, P. Towards an efficient use of the BLAS library for multilinear tensor contractions. *Applied Mathematics and Computation 235* (2014), 454 – 468.

[57] DYNARETEAM. Dynare++: A platform for handling a wide class of economic models (Version 4.5.5). Available from `https://github.com/DynareTeam/dynare`, June 2018.

[58] DYSART, T., KOGGE, P., DENEROFF, M., BOVELL, E., BRIGGS, P., BROCKMAN, J., JACOBSEN, K., JUAN, Y., KUNTZ, S., LETHIN, R., MCMAHON, J., PAWAR, C., PERRIGO, M., RUCKER, S., RUTTENBERG, J., RUTTENBERG, M., AND STEIN, S. Highly scalable near memory processing with migrating threads on the Emu system architecture. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms* (Piscataway, NJ, USA, 2016), IAĴ '16, IEEE Press, pp. 2–9.

[59] EPIFANOVSKY, E., WORMIT, M., KU, T., LANDAU, A., ZUEV, D., KHISTYAEV, K., KALIMAN, I., MANOHAR, P., DREUW, A., AND KRYLOV, A. libtensor (Version 2.5). Available from `http://iopenshell.usc.edu/downloads/tensor`, May 2016.

[60] EPIFANOVSKY, E., WORMIT, M., KU, T., LANDAU, A., ZUEV, D., KHISTYAEV, K., MANOHAR, P., KALIMAN, I., DREUW, A., AND KRYLOV, A. I. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of Computational Chemistry 34*, 26 (2013), 2293–2309.

[61] FARIAS, M. M., PEDROSO, D. M., AND NAKAI, T. Automatic substepping integration of the subloading tij model with stress path dependent hardening. *Computers and Geotechnics 36*, 4 (2009), 537 – 548.

[62] GARCIA, R., AND LUMSDAINE, A. MultiArray: A C++ library for generic programming with arrays. *Software Practive Experience 35* (2004), 159–188.

[63] GARCIA, R., SIEK, J., AND LUMSDAINE, A. Boost.MultiArray from Boost C++ libraries (Version 1.67.0). Available from `https://www.boost.org`, April 2018.

[64] GÖRLITZ, O., SIZOV, S., AND STAAB, S. PINTS: Peer-to-peer infrastructure for tagging systems. In *Proceedings of the 7th International Conference on Peer-to-peer Systems* (Berkeley, CA, USA, 2008), IPTPS'08, USENIX Association, pp. 19–19.

[65] GOTO, K., AND VAN DE GEIJN, R. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw. 35*, 1 (July 2008), 4:1–4:14.

[66] GOTO, K., AND VAN DE GEIJN, R. A. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw. 34*, 3 (May 2008), 12:1–12:25.

[67] GOURVENEC, S., TOMASI, G., DURVILLE, C., DI CRESCENZO, E., SABY, C., MASSART, D. L., BRO, R., AND OPPENHEIM, G. CuBatch, a MATLAB interface for n-mode data analysis. *Chemometrics and Intelligent Laboratory Systems 77*, 1-2 (May 2005), 122–130.

[68] GOURVENEC, S., TOMASI, G., DURVILLE, C., DI CRESCENZO, E., SABY, C., MASSART, D. L., BRO, R., AND OPPENHEIM, G. CuBatch (Version 2.1). Available from `http://www.models.life.ku.dk/cubatch`, 2011.

[69] GRASEDYCK, L. Hierarchical singular value decomposition of tensors. *SIAM J. Matrix Anal. Appl. 31*, 4 (May 2010), 2029–2054.

[70] GRASEDYCK, L., KRESSNER, D., AND TOBLER, C. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen 36*, 1 (2013), 53–78.

[71] HAN, H., AND TSENG, C.-W. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems 17*, 7 (July 2006), 606–618.

[72] HAQUE, S. A., AND HOSSAIN, S. A note on the performance of sparse matrix-vector multiplication with column reordering. In *2009 International Conference on Computing, Engineering and Information* (April 2009), pp. 23–26.

[73] HARSHMAN, R. A. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA Working Papers in Phonetics 16*, 1 (1970), 84.

[74] HEIN, E., CONTE, T., YOUNG, J. S., ESWAR, S., LI, J., LAVIN, P., VUDUC, R., AND RIEDY, J. An initial characterization of the Emu Chick. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops* (May 2018).

[75] HO, J. C., GHOSH, J., AND SUN, J. Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2014), KDD '14, ACM, pp. 115–124.

[76] HOCKNEY, R. W., AND CURINGTON, I. J. $f_{\frac{1}{2}}$: A parameter to characterize memory and communication bottlenecks. *Parallel Computing 10* (1989), 277–286.

[77] JEON, I., PAPALEXAKIS, E. E., KANG, U., AND FALOUTSOS, C. HaTen2: Billion-scale tensor decompositions. In *IEEE International Conference on Data Engineering (ICDE)* (2015).

[78] JEON, I., PAPALEXAKIS, E. E., AND U KANG, C. F. HaTen2: Billion-scale tensor decompositions (Version 1.0). Available from `http://datalab.snu.ac.kr/haten2/`, 2015.

[79] KAMENIK, O. Multidimensional tensor library: algorithms and documentation, 2004. www.cepremap.cnrs.fr/dynare.

[80] KANG, U., PAPALEXAKIS, E., HARPALE, A., AND FALOUTSOS, C. GigaTensor: Scaling tensor analysis up by 100 times - algorithms and discoveries. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2012), KDD '12, ACM, pp. 316–324.

[81] KAO, Y.-J., HSIEH, Y.-D., AND CHEN, P. Uni10: an open-source library for tensor network algorithms. *Journal of Physics: Conference Series 640*, 1 (2015), 012040.

[82] KARSAVURAN, M. O., AKBUDAK, K., AND AYKANAT, C. Locality-aware parallel sparse matrix-vector and matrix-transpose-vector multiplication on many-core processors. *IEEE Transactions on Parallel and Distributed Systems 27*, 6 (June 2016), 1713–1726.

[83] KARYPIS, G., AND KUMAR, V. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing 48*, 1 (1998), 71 – 95.

[84] KAYA, O., AND UÇAR, B. High-performance parallel algorithms for the Tucker decomposition of higher order sparse tensors. Tech. rep., Inria - Research Centre Grenoble Rhone-Alpes, 2015.

[85] KAYA, O., AND UÇAR, B. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 77:1–77:11.

[86] KAYA, O., AND UAR, B. Parallel Candecomp/Parafac decomposition of sparse tensors using dimension trees. *SIAM Journal on Scientific Computing 40*, 1 (2018), C99–C130.

[87] KJOLSTAD, F., KAMIL, S., CHOU, S., LUGATO, D., AND AMARASINGHE, S. The tensor algebra compiler. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017), 77:1–77:29.

[88] KOANANTAKOOL, P., AZAD, A., BULUÇ, A., MOROZOV, D., OH, S. Y., OLIKER, L., AND YELICK, K. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2016), pp. 842–853.

[89] KOLDA, T., AND BADER, B. Tensor decompositions and applications. *SIAM Review 51*, 3 (2009), 455–500.

[90] KOLDA, T. G. Multilinear operators for higher-order decompositions. *Technical Report* (2006).

[91] KOLDA, T. G., AND SUN, J. Scalable tensor decompositions for multi-aspect data mining. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining* (Washington, DC, USA, 2008), ICDM '08, IEEE Computer Society, pp. 363–372.

[92] KOSSAIFI, J., PANAGAKIS, Y., ANANDKUMAR, A., AND PANTIC, M. TensorLy: Tensor learning in Python. *CoRR abs/1610.09555* (2018).

[93] KOURTIS, K., KARAKASIS, V., GOUMAS, G., AND KOZIRIS, N. CSX: An extended compression format for SpMV on shared memory systems. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2011), PPoPP '11, ACM, pp. 247–256.

[94] KRESSNER, D., AND TOBLER, C. Hierarchical Tucker Toolbox (Version 1.2). Available from `http://anchp.epfl.ch/htucker`, Feb 2013.

[95] KRESSNER, D., AND TOBLER, C. Algorithm 941: Htucker—a Matlab Toolbox for tensors in hierarchical Tucker format. *ACM Trans. Math. Softw. 40*, 3 (Apr. 2014), 22:1–22:22.

[96] LANDRY, W. Implementing a high performance tensor library. *Sci. Program. 11*, 4 (Dec. 2003), 273–290.

[97] LANDRY, W. FTensor. Available from `http://www.wlandry.net/Projects/FTensor`, April 2018.

[98] LATCHOUMANE, C.-F. V., VIALATTE, F.-B., SOLÉ-CASALS, J., MAURICE, M., WIMALARATNA, S. R., HUDSON, N., JEONG, J., AND CICHOCKI, A. Multiway array decomposition analysis of EEGs in Alzheimer's disease. *Journal of neuroscience methods 207*, 1 (2012), 41–50.

[99] LEBEDEV, V., GANIN, Y., RAKHUBA, M., OSELEDETS, I., AND LEMPITSKY, V. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. *arXiv preprint arXiv:1412.6553* (2014).

[100] LI, J., BATTAGLINO, C., PERROS, I., SUN, J., AND VUDUC, R. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *ACM/IEEE Supercomputing (SC '15)* (New York, NY, USA, 2015), ACM.

[101] LI, J., CHOI, J., PERROS, I., SUN, J., AND VUDUC, R. Model-driven sparse CP decomposition for higher-order tensors. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2017), pp. 1048–1057.

[102] LI, J., MA, Y., AND VUDUC, R. ParTI!: A Parallel Tensor Infrastructure for Multicore CPU and GPUs (Version 0.1.0). Available from `https://github.com/hpcgarage/ParTI`, 2016.

[103] LI, J., MA, Y., YAN, C., AND VUDUC, R. Optimizing sparse tensor times matrix on multi-core and many-core architectures. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms* (Piscataway, NJ, USA, 2016), IAˆ3 '16, IEEE Press, pp. 26–33.

[104] LI, J., SUN, J., AND VUDUC, R. HiCOO: Hierarchical storage of sparse tensors. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)* (Dallas, TX, USA, November 2018). (*to appear*).

[105] LI, J., TAN, G., CHEN, M., AND SUN, N. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 117–126.

[106] LIMACHE, A. C., AND ROJAS FREDINI, P. S. LTensor: A high performance C++ tensor library based on index notation (Version 21112012). Available from `http://code.google.com/p/ltensor/`, Nov 2012.

[107] LIU, B., WEN, C., SARWATE, A. D., AND DEHNAVI, M. M. A unified optimization approach for sparse tensor operations on GPUs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)* (Sept 2017), pp. 47–57.

[108] LIU, W., AND VINTER, B. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2014), IPDPS '14, IEEE Computer Society, pp. 370–381.

[109] LIU, W., AND VINTER, B. A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. *Journal of Parallel and Distributed Computing 85*, C (nov 2015), 47–61.

[110] LIU, W., AND VINTER, B. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM International Conference on Supercomputing* (2015), ICS '15, ACM, pp. 339–350.

[111] LU, H., PLATANIOTIS, K. N., AND VENETSANOPOULOS, A. N. A survey of multilinear subspace learning for tensor data. *Pattern Recognition 44*, 7 (2011), 1540 – 1551.

[112] LUBIW, A. Doubly lexical orderings of matrices. *SIAM Journal on Computing 16*, 5 (1987), 854–879.

[113] MA, Y., LI, J., WU, X., YAN, C., SUN, J., AND VUDUC, R. Optimizing sparse tensor times matrix on GPUs. *Journal of Parallel and Distributed Computing* (2018). (*To appear*).

[114] MCCALPIN, J. Memory bandwidth and machine balance in high performance computers. 19–25.

[115] MELHEM, R. G., AND RAMARAO, K. V. S. Multicolor reordering of sparse matrices resulting from irregular grids. *ACM Trans. Math. Softw. 14*, 2 (June 1988), 117–138.

[116] MELLOR-CRUMMEY, J., WHALLEY, D., AND KENNEDY, K. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming 29*, 3 (Jun 2001), 217–247.

[117] MERRILL, D., AND GARLAND, M. Merge-based parallel sparse matrix-vector multiplication. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2016), pp. 678–689.

[118] MILES, E., WHITE, S. R., ET AL. ITensor, (Version 2.1.1). Available from `http://itensor.org`, Aug 2017.

[119] MORTON, G. M. A computer oriented geodetic data base; and a new technique in file sequencing. Tech. rep., Ottawa, Canada: IBM Ltd, 1966.

[120] NAGASAKA, Y., NUKADA, A., AND MATSUOKA, S. High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)* (Aug 2017), pp. 101–110.

[121] ND OTHERS, Y.-D. H. Uni10: The universal tensor network library (Version 2.0.0). Available from `http://www.uni10.org`, Jan 2018.

[122] NICKEL, M. scikit-tensor library (Version 0.1). Available from `https://pypi.python.org/pypi/scikit-tensor`, Feb 2014.

[123] NOVIKOV, A., PODOPRIKHIN, D., OSOKIN, A., AND VETROV, D. Tensorizing neural networks. *CoRR abs/1509.06569* (2015).

[124] OLIPHANT, T. E. Python for scientific computing. *Computing in Science Engineering 9*, 3 (May 2007), 10–20.

[125] OSELEDETS, I., ET AL. TT-Toolbox (Version 2.3). Available from `https://github.com/oseledets/TT-Toolbox`, 2014.

[126] OSELEDETS, I. V. Tensor-train decomposition. *SIAM J. Sci. Comput. 33*, 5 (Sept. 2011), 2295–2317.

[127] OZOG, D., HAMMOND, J. R., DINAN, J., BALAJI, P., SHENDE, S., AND MALONY, A. Inspector-executor load balancing algorithms for block-sparse tensor contractions. In *Parallel Processing (ICPP), 2013 42nd International Conference on* (Oct 2013), pp. 30–39.

[128] PAIGE, R., AND TARJAN, R. E. Three partition refinement algorithms. *SIAM Journal on Computing 16*, 6 (Dec. 1987), 973–989.

[129] PAPALEXAKIS, E. E., FALOUTSOS, C., AND SIDIROPOULOS, N. D. ParCube: Sparse parallelizable tensor decompositions. In *Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I* (Berlin, Heidelberg, 2012), ECML PKDD'12, Springer-Verlag, pp. 521–536.

[130] PAPALEXAKIS, E. E., FALOUTSOS, C., AND SIDIROPOULOS, N. D. ParCube: Sparse parallelizable CANDECOMP-PARAFAC tensor decomposition. *ACM Trans. Knowl. Discov. Data 10*, 1 (July 2015), 3:1–3:25.

[131] PAPALEXAKIS, E. E., KANG, U., FALOUTSOS, C., SIDIROPOULOS, N. D., AND HARPALE, A. Large scale tensor decompositions: Algorithmic developments and applications. *IEEE Data Eng. Bull. 36*, 3 (2013), 59–66.

[132] PEDROSO, D. Tensors C++ library for tensor algebra (Version 1.0). Available from `http://www.nongnu.org/tensors/index.shtml`, 2012.

[133] PEDROSO, D. M., FARIAS, M. M., AND NAKAI, T. An interpretation of subloading tij model in the context of conventional elastoplasticity theory. *Soils and Foundations 45*, 4 (2005), 61–77.

[134] PERROS, I., CHEN, R., VUDUC, R., AND SUN, J. Sparse hierarchical Tucker factorization and its application to healthcare. *IEEE International Conference on Data Mining (ICDM)* (2015).

[135] PERROS, I., PAPALEXAKIS, E. E., WANG, F., VUDUC, R., SEARLES, E., THOMPSON, M., AND SUN, J. SPARTan: Scalable PARAFAC2 for large & sparse data. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2017), KDD '17, ACM, pp. 375–384.

[136] PHAN, A.-H., TICHAVSKY, P., AND CICHOCKI, A. TENSORBOX: a Matlab package for tensor decomposition (Version 2015.11). Available from `http://www.bsp.brain.riken.jp/~phan/tensorbox.php`, Nov 2015.

[137] PHAN, A. H., TICHAVSK, P., AND CICHOCKI, A. Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations. *IEEE Transactions on Signal Processing 61*, 19 (Oct 2013), 4834–4846.

[138] PHIPPS, E. T., KOLDA, T. G., DUNLAVY, D., BALLARD, G., AND PLANTENGA, T. Genten: Software for generalized tensor decompositions v. 1.0.0, 6 2017.

[139] PICHEL, J., HERAS, D., CABALEIRO, J., AND RIVERA, F. Performance optimization of irregular codes based on the combination of reordering and blocking techniques. *Parallel Computing 31*, 8 (2005), 858 – 876.

[140] PICHEL, J. C., RIVERA, F. F., FERNNDEZ, M., AND RODRGUEZ, A. Optimization of sparse matrixvector multiplication using reordering techniques on GPUs. *Microprocessors and Microsystems 36*, 2 (2012), 65 – 77. SPECIAL ISSUE - EXPLOITATION OF HARDWARE ACCELERATORS.

[141] PICHEL, J. C., SINGH, D. E., AND CARRETERO, J. Reordering algorithms for increasing locality on multicore processors. In *2008 10th IEEE International Conference on High Performance Computing and Communications* (Sept 2008), pp. 123–130.

[142] POTHEN, A., AND FAN, C.-J. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw. 16*, 4 (Dec. 1990), 303–324.

[143] RAMANATHAN, A., AGARWAL, P. K., KURNIKOVA, M., AND LANGMEAD, C. J. *An online approach for mining collective behaviors from molecular dynamics simulations*, vol. LNCS 5541. 2009, pp. pp. 138–154.

[144] RAMANATHAN, A., YOO, J. O., AND LANGMEAD, C. J. PyTensor (Version r2). Available from `https://code.google.com/p/pytensor/`, 2009.

[145] RAMANATHAN, A., YOO, J. O., AND LANGMEAD, C. J. On-the-fly identification of conformational substates from molecular dynamics simulations. *Journal of Chemical Theory and Computation 7*, 3 (2011), 778–789. PMID: 26596308.

[146] RAVINDRAN, N., SIDIROPOULOS, N. D., SMITH, S., AND KARYPIS, G. Memory-efficient parallel computation of tensor and matrix products for big tensor decompositions. *Proceedings of the Asilomar Conference on Signals, Systems, and Computers* (2014).

[147] SAAD, Y. SPARSKIT: a basic tool kit for sparse matrix computations, 1990.

[148] SANDERS, B., BARTLETT, R., DEUMENS, E., LOTRICH, V., AND PONTON, M. A block-oriented language and runtime system for tensor algebra with very large arrays. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for* (Nov 2010), pp. 1–11.

[149] SCHATZ, M. D. Anatomy of parallel computation with tensors. *The University of Texas at Austin, Department of Computer Science Report* (Dec. 2013).

[150] SIDIROPOULOS, N. D., DE LATHAUWER, L., FU, X., HUANG, K., PAPALEXAKIS, E. E., AND FALOUTSOS, C. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on Signal Processing 65*, 13 (July 2017), 3551–3582.

[151] SMITH, S., CHOI, J. W., LI, J., VUDUC, R., PARK, J., LIU, X., AND KARYPIS, G. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools, 2017.

[152] SMITH, S., AND KARYPIS, G. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms* (2015), ACM, p. 7.

[153] SMITH, S., AND KARYPIS, G. A medium-grained algorithm for distributed sparse tensor factorization. In *Parallel and Distributed Processing Symposium (IPDPS), 2016 IEEE International* (2016), IEEE.

[154] SMITH, S., AND KARYPIS, G. Accelerating the Tucker decomposition with compressed sparse tensors. In *European Conference on Parallel Processing* (2017), Springer.

[155] SMITH, S., PARK, J., AND KARYPIS, G. Sparse tensor factorization on many-core processors with high-bandwidth memory. *31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17)* (2017).

[156] SMITH, S., RAVINDRAN, N., SIDIROPOULOS, N., AND KARYPIS, G. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium* (2015), IPDPS.

[157] SMITH, S., RAVINDRAN, N., SIDIROPOULOS, N., AND KARYPIS, G. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit (Version 1.1.1). Available from `https://github.com/ShadenSmith/splatt`, 2016.

[158] SOLOMONIK, E. *Provably Efficient Algorithms for Numerical Tensor Algebra*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2014.

[159] SOLOMONIK, E., DEMMEL, J., AND HOEFLER, T. Communication lower bounds for tensor contraction algorithms. Tech. rep., ETH Zürich, 2015.

[160] SOLOMONIK, E., ET AL. Cyclops Tensor Framework (Version 1.5.2). Available from `https://github.com/cyclops-community/ctf`, Jun. 2018.

[161] SOLOMONIK, E., AND HOEFLER, T. Sparse Tensor Algebra as a Parallel Programming Model. *ArXiv e-prints* (Nov. 2015).

[162] SOLOMONIK, E., MATTHEWS, D., HAMMOND, J., AND DEMMEL, J. Cyclops Tensor Framework: reducing communication and eliminating load imbalance in massively parallel contractions. Tech. Rep. UCB/EECS-2012-210, EECS Department, University of California, Berkeley, Nov 2012.

[163] SOLOMONIK, E., MATTHEWS, D., HAMMOND, J. R., STANTON, J. F., AND DEMMEL, J. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing 74*, 12 (2014), 3176–3190.

[164] TARJAN, R. E., AND YANNAKAKIS, M. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing 13*, 3 (1984), 566–579.

[165] TUCKER, L. R. Some mathematical notes on three-mode factor analysis. *Psychometrika 31* (1966), 279–311.

[166] VALEEV, E., NAKATANI, N., STOUDENMIRE, M., SHIOZAKI, T., AND CHAN, G. Basic Tensor Algebra Subroutines (Version v0.0.1). Available from `https://github.com/BTAS/BTAS/`, 2018.

[167] VAN DER WALT, S., COLBERT, S., AND VAROQUAUX, G. The NumPy array: A structure for efficient numerical computation. *Computing in Science Engineering 13*, 2 (March 2011), 22–30.

[168] VAN ZEE, F. G., AND VAN DE GEIJN, R. A. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw. 41*, 3 (June 2015), 14:1–14:33.

[169] VASILESCU, M. A. O., AND TERZOPOULOS, D. Multilinear analysis of image ensembles: TensorFaces. In *IN PROCEEDINGS OF THE EUROPEAN CONFERENCE ON COMPUTER VISION* (2002), pp. 447–460.

[170] VELDHUIZEN, T., ET AL. The Blitz++ library (Version 0.10). Available from `http://sourceforge.net/projects/blitz/`, March 2014.

[171] VERVLIET, N., DEBALS, O., SORBER, L., VAN BAREL, M., AND DE LATHAUWER, L. Tensorlab (Version 3.0). Available from `http://www.tensorlab.net`, March 2016.

[172] VUDUC, R., DEMMEL, J. W., AND YELICK, K. A. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series 16*, 1 (2005), 521.

[173] VUDUC, R. W. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, CA, USA, January 2004.

[174] WANG, H., LIU, W., HOU, K., AND FENG, W.-C. Parallel transposition of sparse data structures. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), ICS '16, pp. 33:1–33:13.

[175] WANG, Y., CHEN, R., GHOSH, J., DENNY, J. C., KHO, A., CHEN, Y., MALIN, B. A., AND SUN, J. Rubik: Knowledge guided tensor factorization and completion for health data analytics. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2015), KDD '15, ACM, pp. 1265–1274.

[176] WHALEY, R. C., AND DONGARRA, J. J. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 1998), SC '98, IEEE Computer Society, pp. 1–27.

[177] WILLCOCK, J., AND LUMSDAINE, A. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th Annual International Conference on Supercomputing* (New York, NY, USA, 2006), ICS '06, ACM, pp. 307–316.

[178] WILLIAMS, S., OLIKER, L., VUDUC, R., SHALF, J., YELICK, K., AND DEMMEL, J. Optimization of sparse matrixvector multiplication on emerging multicore platforms. *Parallel Computing 35*, 3 (2009), 178 – 194. Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[179] WISE, B. M., AND GALLAGHER, N. B. PLS_Toolbox (Version 8.6.2). Available from `http://www.eigenvector.com/software/index.htm`, Jun. 2018.

[180] XIE, B., ZHAN, J., LIU, X., GAO, W., JIA, Z., HE, X., AND ZHANG, L. CVR: Efficient vectorization of SpMV on x86 processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (New York, NY, USA, 2018), CGO 2018, ACM, pp. 149–162.

[181] YAN, S., LI, C., ZHANG, Y., AND ZHOU, H. yaSpMV: Yet another SpMV framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2014), PPoPP '14, ACM, pp. 107–118.

[182] YANG, C., BULUÇ, A., AND OWENS, J. Design principles for sparse matrix multiplication on the GPU. *24th International European Conference on Parallel and Distributed Computing* (2018).

[183] YOO, J. O., RAMANATHAN, A., AND LANGMEAD, C. J. PyTensor: A Python based tensor library. Tech. rep., CMU, 2010.

[184] YZELMAN, A. J. N., AND ROOSE, D. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems 25*, 1 (Jan 2014), 116–125.

[185] YZELMAN, A. N. *Fast sparse matrix-vector multiplication by partitioning and reordering*. PhD thesis, Utrecht University, Utrecht, the Netherlands, October 2011.

[186] ZHANG, X. OpenBLAS : An optimized BLAS library.

[187] ZHAO, Y., LI, J., LIAO, C., AND SHEN, X. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2018), PPoPP '18, ACM, pp. 94–108.

[188] ZHOU, G., AND CICHOCKI, A. A brief guide for TDALAB, April 2013.

[189] ZHOU, G., AND CICHOCKI, A. Matlab Toolbox for tensor decomposition & analysis (Version 1.1). Available from `http://www.bsp.brain.riken.jp/TDALAB/`, May 2013.

[190] ZHOU, S., VINH, N. X., BAILEY, J., JIA, Y., AND DAVIDSON, I. Accelerating online CP decompositions for higher order tensors. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2016), KDD '16, ACM, pp. 1375–1384.

VITA

Jiajia Li is a 5th-year Ph.D. candidate in Computational Science & Engineering at Georgia Institute of Technology. Before, she was a research intern of IBM Thomas J. Watson Research Center and Intel Parallel Computing Lab in the summers of 2015 and 2016 respectively. In the past, she received a Ph.D. degree from Institute of Computing Technology at Chinese Academy of Sciences. Her current research emphasizes on optimizing tensor algorithms including tensor decompositions and fundamental tensor operations especially for sparse data by utilizing various parallel architectures. She has published in top-tier high performance computing and other related venues, such as SC, ICS, PLDI, IPDPS, etc.